vacuumlabs

FluidTokens Lending v3

Audit Report v1

August 27, 2025

Contents

Revision table

Report version	Report name	Date	Report URL
1.0	Main audit	2025-08-27	Full report link

1 Executive summary

This report does not provide any warranty of quality or security of the audited code and should be understood as a best efforts opinion of Vacuumlabs produced upon reviewing the materials provided to Vacuumlabs. Vacuumlabs can only comment on the issues it discovers and Vacuumlabs does not guarantee discovering all the relevant issues. Vacuumlabs also disclaims all warranties or guarantees in relation to the report to the maximum extent permitted by the applicable law. This report is also subject to the full disclaimer in the appendix of this document, which you should read before reading the report.

Project overview

FluidTokens Lending v3 is a **highly customizable decentralized lending platform** that supports both peer-to-peer lending and a kind of single-party liquidity pool-based lending on Cardano. The protocol enables flexible loan arrangements where borrowers can create requests specifying their desired loan terms, while lenders can either accept these requests or create their own lending pools from which multiple borrowers can take loans.

The platform supports **two primary lending mechanisms**: static fixed loans with no price adjustments throughout the term, and dynamic loans where borrowable amounts are determined dynamically by loan-to-value ratios and prices are determined by oracle-based price feeds. A key feature is the ability for borrowers to aggregate loans from multiple pools within a single transaction, providing significant flexibility in sourcing the required capital. It is important to note that lending pools refer to liquidity provided by a single party, not multiple parties pooling funds together as in some other DeFi protocols. Once a loan is taken from a pool, it is a simple peer-to-peer loan between the borrower and the lender.

Collateral management is comprehensive, supporting both native tokens, NFTs and CIP-113 programmable tokens (see more below), with each loan from a pool limited to a single collateral type. The protocol includes sophisticated **loan term structures** with three distinct repayment modes:

- **Interest on Remaining Principal**: Uses full amortization formulae where each installment includes both principal and interest calculated on the remaining balance. There is a constant installment amount.

- **Principal and Interest on Installments**: Total interest is calculated upfront and divided evenly across installments.
- **Perpetual Loans**: The loan can be taken for indefinitely. Borrowers pay only part of the interest in installments with additional optional linear interest rate increases over time. The principal remains constant and the whole outstanding debt is paid off at the end of the loan. The debt increases and thus the collateral value must increase as well, to still sufficiently cover the debt.

All loan types support configurable grace periods, late payment penalties, liquidation discounts and flexible installment scheduling.

Loan recasting is optionally available for both Interest on Remaining Principal and Perpetual Loan modes, allowing borrowers to make additional principal payments to reduce future installment obligations. Recasting is subject to restrictions including the requirement that all due installments plus one additional installment must be paid before recasting is permitted. In perpetual loans, the remaining interest that is not part of installments needs to be also fully paid off — this can be done directly in the recast transaction. There are configurable limits on the maximum number of recasts allowed per loan.

Liquidation mechanisms offer four distinct approaches: no liquidation with lenders claiming full collateral upon default, no liquidation with Dutch auction systems that can return excess value to borrowers on a who-buys-highest basis, oracle-based liquidation with full collateral forfeiture, and oracle-based liquidation with partial liquidation where borrowers receive compensation for collateral value exceeding the debt less the liquidation discount. The protocol also supports **partial liquidations** where only the necessary portion of collateral plus a liquidation discount is claimed to cover outstanding obligations.

Position management is handled through bond tokens minted for both lenders and borrowers, enabling the transfer of rights and responsibilities through token ownership. Advanced features include refinancing capabilities and permissioned lending with KYC token requirements for compliance. Note: The KYC token and its binding to users' wallets is outside the scope of this audit. Even though the token's presence is checked when lending from a permissioned pool, nothing is checked when the bond token and thus the position is transferred.

Oracle integration supports two main price feed types:

- **In-house oracles**: Aggregated feeds from sources such as centralized exchanges and dedicated feeds for specialized pricing requirements. The price logistics are outside of the scope of this audit. The on-chain fully trusts the prices provided by valid oracles and does not check them.

- **Third-party integrations**: Support for external oracle providers including Orcfax and Charli3. The protocol has means of not allowing some oracle types to be used for some assets, even though this is hardcoded and can not be easily updated once a loan is taken.

The protocol implements full **CIP-113 programmable token support**, allowing seamless integration with smart tokens that require additional validation logic during transfers. A significant portion of the protocol logic is designed to work with programmable tokens, handling both traditional addresses and programmable token credentials throughout the system seamlessly. Note that this support is novel and users should be super-cautious of the types of tokens they are borrowing or lending, as the tokens can be programmable and malicious programmable tokens can block execution and lead to unforeseen consequences s.a. repayment impossibility.

The protocol's technical architecture centers around multiple interconnected validators mostly using transaction-level validation through the withdraw-zero trick, each checking all their delegated inputs found on either a general spend script referring the validation path, or a programmable credential referring the validation path (CIP-113 programmable tokens need to be locked at the smart wallet script). The reference is either hardcoded in the general spend script or noted in the staking credential in the smart tokens' case. Overall, this creates a complex but flexible system with multiple extensions possible in the future.

Important considerations of the protocol include:

- **Oracle feeds**: Oracle data can not be revoked once published, so validity periods should be kept appropriately short. Additionally, the prices are trusted by the protocol and the logistics of obtaining the prices is outside of the scope of this audit.
- **CIP-113 programmable tokens**: This protocol represents one of the first, probably the very first, production implementations using CIP-113 programmable tokens on Cardano. The programmable tokens as well as smart wallet code are outside of the scope of this audit. However, they constitute a significant part of the security of the protocol.
- **Data sanitization**: The audit assumes that loan data presented to users through interfaces is properly sanitized and users understand what loan terms they are signing for by accepting the terms. Users should thoroughly examine loan terms and use trusted interfaces.
- **Protocol configuration**: There is a protocol-level configuration locked at a single UTxO that is referenced from all the scripts. There is an **admin credential** — not

necessarily a single key — that has the ultimate power over the protocol as it can update some key properties s.a. the hashes of the respective protocol scripts, leading to potentially catastrophic consequences if misused.

Audit overview

I started the audit at commit `91c813888e3d4a9cca0aefabd4db6e8464c00aa7` and it lasted from January 24, 2025 to August 27, 2025. The timeframe included additional feature additions, significant refactors, and periods during which I awaited the implementation of fixes by the client. We primarily interacted through Discord and provided feedback via continuous reporting.

The scope of the audit was limited to the smart contract files only. I did not review any tests as part of this audit. I performed a design review along with a deep manual audit of the code and reported findings along with remediation suggestions to the team in a continuous fashion, allowing time for proper remediation that I reviewed afterwards. See more about our methodology in Methodology.

The commit `aed7d340119c56e8f8f02cbefcc53810ce7df0fc` represents the final version of the code. The status of any issue in this report reflects its status at the most recent reviewed commit. You can see all the files audited and their hashes in Audited files. The smart contract language used is Aiken and the contracts are intended to run on Cardano. To avoid any doubt, I did not audit Aiken itself, the underlying CIP-113 programmable token implementation or 3rd party oracle providers' implementations.

Summary of findings

During the audit, I identified and reported 33 critical, 6 major, 8 medium, 18 minor, and 4 informational findings. For details on severity and status classification, please refer to Classification.

All findings have been resolved except for three minor findings and one informational finding that were acknowledged as acceptable design decisions:

- **FTL3-309**: Oracle safe-guards suggestion (Minor, Acknowledged). This finding concerns potential oracle manipulation risks depending on the actual source of prices. The client acknowledged the concern and plans to use Charli3 and highly liquid assets to mitigate these risks.
- **FTL3-314**: Installment amounts might not add up to the total principal and interest (Minor, Acknowledged). Due to rounding in individual installment calculations, the

total repaid amount might slightly exceed the theoretical total. This was acknowledged as an acceptable trade-off that can be handled through clear UX communication.

- **FTL3-317**: Native tokens can be sent to programmable credential (Minor, Acknowledged). While native tokens might inadvertently be sent to programmable token credentials causing mild inconvenience, they can be retrieved, so this was acknowledged as acceptable.
- **FTL3-401**: Equity payment is in the principal asset (Informational, Acknowledged). The protocol returns liquidation equity in the principal asset rather than the collateral asset, which differs from most other protocols but was acknowledged as an acceptable design choice.

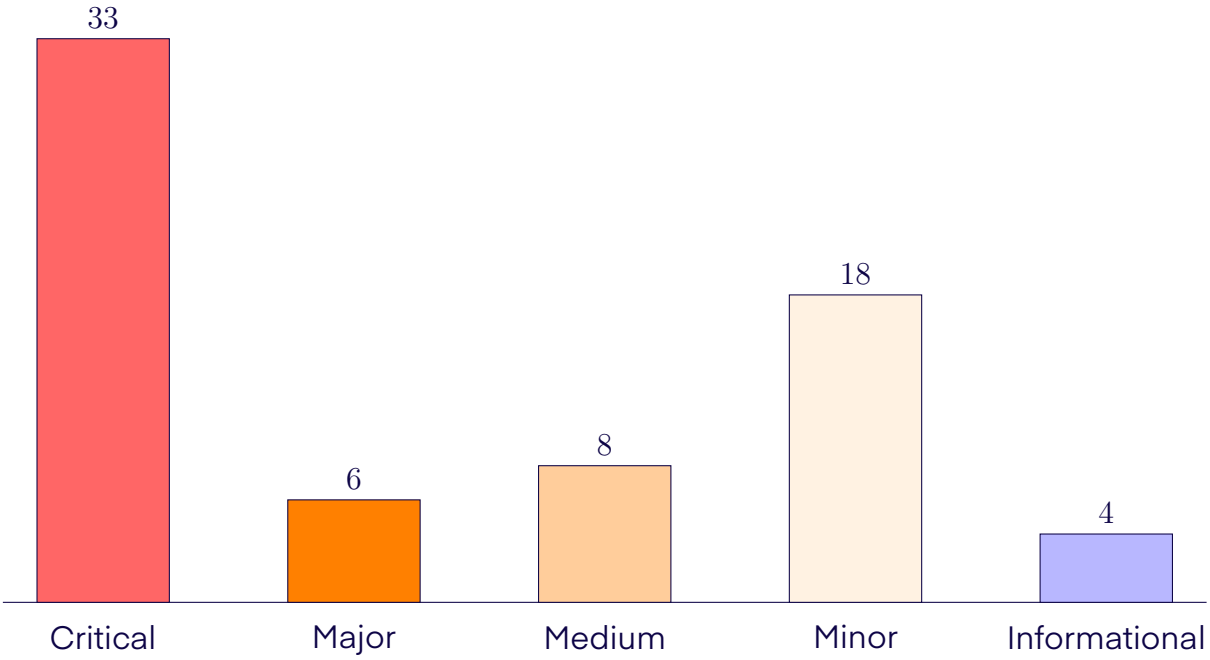
The reported findings can be broadly categorized as follows:

- **Fundamental protocol breakage:** Issues that completely prevented core functionality from operating, including missing token burning logic in minting policies (FTL3-001, FTL3-003), broken programmable token support due to incorrect credential lookups (FTL3-009, FTL3-010, FTL3-011), and configuration parsing failures (FTL3-030, FTL3-031). Many of these issues would have been caught by comprehensive happy-path testing scenarios.
- **Economic attacks and fund theft:** Vulnerabilities allowing malicious actors to steal funds or manipulate the system, including borrowers stealing lender pool funds through address manipulation (FTL3-002), lenders claiming entire collateral amounts in Dutch auctions through malicious configuration (FTL3-004, FTL3-005, FTL3-012), programmable tokens being weaponized to block or ransom protocol operations (FTL3-006), and incorrect field updates in repayment logic allowing lenders to claim collateral from borrowers who have actually repaid (FTL3-033).
- **Logical vulnerabilities and mathematical errors:** Critical flaws in business logic including completely reversed liquidation threshold comparisons (FTL3-015), incorrect amortization formulas leading to massive overpayments (FTL3-016), flawed recasting logic (FTL3-017, FTL3-018), inconsistent interest calculations (FTL3-019, FTL3-020), and LTV calculations based only on initial principal rather than remaining debt (FTL3-032).
- **Technical implementation issues:** Problems with hash function limitations causing transaction failures (FTL3-008), address validation bypasses allowing bond-based attacks (FTL3-028, FTL3-029), and programmable token-specific validation bugs (FTL3-023, FTL3-024, FTL3-025, FTL3-026). Many issues involved indexing problems throughout the codebase.

- **Oracle manipulation and cross-protocol attacks:** Issues with hardcoded DEX fees providing incorrect pricing (FTL3-101), double-satisfaction attacks where single payments fulfill multiple obligations (FTL3-102, FTL3-104), and AMM formula errors in oracle price calculations (FTL3-013.
- **Denial of service and operational issues:** Various attack vectors allowing malicious parties to block normal operations, including transaction manipulation (FTL3-008), bond-based DoS attacks (FTL3-028), and edge cases in liquidation mechanics (FTL3-207, FTL3-208).
- **Code quality and documentation:** Non-critical improvements for clarity and maintainability, including naming inconsistencies, unused functions, and documentation gaps (FTL3-402). The codebase extensively uses fields with dummy values rather than proper optional types.

Overall, the remediation efforts have dramatically improved FluidTokens Lending v3's security posture. The complexity of the protocol, particularly its advanced features like programmable token support and sophisticated liquidation mechanisms, presented significant challenges and thus the overall number of issues. The collaborative audit process resulted in a more robust and secure lending platform ready for deployment.

2 Severity overview



Findings

ID	TITLE	SEVERITY	STATUS
FTL3-001	Repayments can not be withdrawn	CRITICAL	RESOLVED
FTL3-002	Lender pool funds can be stolen	CRITICAL	RESOLVED
FTL3-003	Collateral can not be withdrawn	CRITICAL	RESOLVED
FTL3-004	Lender can claim the whole collateral in a dutch auction before start	CRITICAL	RESOLVED
FTL3-005	Lender can claim the whole collateral in an auction by malicious address	CRITICAL	RESOLVED
FTL3-006	Blocking funds and gaining unfair advantage by adding a programmable token	CRITICAL	RESOLVED

Continued on next page

ID	TITLE	SEVERITY	STATUS
FTL3-007	Ada collateral is not protected in requests	CRITICAL	RESOLVED
FTL3-008	Lender can disable repaying and liquidate	CRITICAL	RESOLVED
FTL3-009	Loan token can not be minted for programmable token loans	CRITICAL	RESOLVED
FTL3-010	Repayment token can not be minted for programmable token loans	CRITICAL	RESOLVED
FTL3-011	Protocol is unfeasible due to usage of `get_outputs_to_smart_credential`	CRITICAL	RESOLVED
FTL3-012	Dutch auction's borrower compensation goes to the lender	CRITICAL	RESOLVED
FTL3-013	AMM formulas are incorrect	CRITICAL	RESOLVED
FTL3-014	Wrong arguments in conversion from Ada to token	CRITICAL	RESOLVED
FTL3-015	Healthy loans can be liquidated	CRITICAL	RESOLVED
FTL3-016	Amortization formula is wrong	CRITICAL	RESOLVED
FTL3-017	Recasting does not work well with the amortization formula	CRITICAL	RESOLVED
FTL3-018	Recasting on due loan installments avoids interest and penalties	CRITICAL	RESOLVED
FTL3-019	Inconsistent perpetual loan interest computation	CRITICAL	RESOLVED
FTL3-020	Remaining debt on perpetual loans does not assume previous payments	CRITICAL	RESOLVED
FTL3-021	Dutch auction payments can break due to checking bond addresses	CRITICAL	RESOLVED
FTL3-022	Perpetual loan recasting logic is incorrect	CRITICAL	RESOLVED

Continued on next page

ID	TITLE	SEVERITY	STATUS
FTL3-023	Loan inputs with programmable assets bypass action validator checks	CRITICAL	RESOLVED
FTL3-024	Wrong action credential allows borrowers to unlock programmable collateral	CRITICAL	RESOLVED
FTL3-025	Wrong receipt condition allows blocking funds with programmable assets	CRITICAL	RESOLVED
FTL3-026	Programmable collateral sent to uncontrollable auction credential is lost	CRITICAL	RESOLVED
FTL3-027	Zero liquidation penalty incorrectly skips equity return to borrower	CRITICAL	RESOLVED
FTL3-028	Malicious parties can block transactions by holding bonds without stake credentials	CRITICAL	RESOLVED
FTL3-029	Bond address trusted without bond presence verification	CRITICAL	RESOLVED
FTL3-030	Datums can not be parsed	CRITICAL	RESOLVED
FTL3-031	Wrong config index extracts incorrect loan policy id	CRITICAL	RESOLVED
FTL3-032	LTV is calculated based on the initial principal	CRITICAL	RESOLVED
FTL3-033	Repayment increments wrong field causing eventual collateral loss	CRITICAL	RESOLVED
FTL3-101	DEX oracle computation uses hardcoded fees	MAJOR	RESOLVED
FTL3-102	Ada in expired requests is vulnerable to double satisfaction	MAJOR	RESOLVED
FTL3-103	Too big loan can liquidate the borrower	MAJOR	RESOLVED
FTL3-104	Cross-script double satisfaction	MAJOR	RESOLVED
FTL3-105	Permissioned conditions not enforced for programmable tokens	MAJOR	RESOLVED

Continued on next page

ID	TITLE	SEVERITY	STATUS
FTL3-106	Time unit change error disables recasts	MAJOR	RESOLVED
FTL3-201	Minting multiple repayment tokens is nearly unfeasible	MEDIUM	RESOLVED
FTL3-202	Request can not be cancelled after expiration by a different party	MEDIUM	RESOLVED
FTL3-203	It is possible to lend to an expired request	MEDIUM	RESOLVED
FTL3-204	Pool might be blocked until recreated	MEDIUM	RESOLVED
FTL3-205	Too small Ada equity makes liquidation impossible	MEDIUM	RESOLVED
FTL3-206	It might be impossible to add collateral to a non-specific asset collateral loan	MEDIUM	RESOLVED
FTL3-207	No liquidation discount	MEDIUM	RESOLVED
FTL3-208	User stake credentials to authorize programmable token transfers	MEDIUM	RESOLVED
FTL3-301	Permissioned lending party is chosen by an index out of context	MINOR	RESOLVED
FTL3-302	Oracle's `valid_from` is unchecked	MINOR	RESOLVED
FTL3-303	Dutch auction can be bought before it starts	MINOR	RESOLVED
FTL3-304	Indexing repayments in repayment minting policy is troublesome	MINOR	RESOLVED
FTL3-305	Burning and minting request and pool tokens is inconvenient	MINOR	RESOLVED
FTL3-306	The `principalLTV` variable is overused	MINOR	RESOLVED
FTL3-307	Ada oracle use is inconsistent	MINOR	RESOLVED

Continued on next page

ID	TITLE	SEVERITY	STATUS
FTL3-308	AMM formulas are based on a rational number that is then rounded	MINOR	RESOLVED
FTL3-309	Oracle safe-guards suggestion	MINOR	ACKNOWLEDGED
FTL3-310	Equity computation charges conversion fees to the lender	MINOR	RESOLVED
FTL3-311	Pool KYC token signature can be reused to borrow more	MINOR	RESOLVED
FTL3-312	Unlimited recasts do not work	MINOR	RESOLVED
FTL3-313	Borrowers can avoid late repayment penalty	MINOR	RESOLVED
FTL3-314	Installment amounts might not add up to the total principal and interest	MINOR	ACKNOWLEDGED
FTL3-315	Total installments field for perpetual loans	MINOR	RESOLVED
FTL3-316	Hash function mismatch in oracle key verification	MINOR	RESOLVED
FTL3-317	Native tokens can be sent to programmable credential	MINOR	ACKNOWLEDGED
FTL3-318	Big bond reference inputs can cause DoS via transaction limits	MINOR	RESOLVED
FTL3-401	Dropping a byte of a hash result is discouraged	INFORMATIONAL	RESOLVED
FTL3-402	Request id and pool id might be identical	INFORMATIONAL	RESOLVED
FTL3-403	Equity payment is in the principal asset	INFORMATIONAL	ACKNOWLEDGED
FTL3-404	Code quality, naming, and documentation issues	INFORMATIONAL	RESOLVED

FTL3-001 Repayments can not be withdrawn

Category	Vulnerable commit	Severity	Status
Logical Issue	`88d07d5a7c`	CRITICAL	RESOLVED

Description

A repayment token needs to be burned in order to withdraw funds from a repayment. However, the minting policy does not allow any burning to happen. That means that the validator would never succeed and thus no repayment can ever be withdrawn. The logic was intended to be present as evidenced by the minting function's comment. However, it's not there at the moment.

Recommendation

I suggest carefully adding in the burning mechanism.

Resolution

The issue was fixed in the commit `3565170f94`.

FTL3-002 Lender pool funds can be stolen

Category	Vulnerable commit	Severity	Status
Logical Issue	`88d07d5a7c`	CRITICAL	RESOLVED

Description

If a borrower borrows from a lender pool an amount that is less than the total amount that is present there, he is supposed to return the remaining funds in a recreated lender pool. In the validation logic checking this, the lender pool's address is unchecked with a comment explaining why it's okay to not check it there: "outputs have been already filtered at the beginning". In other cases where this comment is present, that is true. However, this specific recreated pool output is not filtered previously. Its address can really be arbitrary.

If the borrower is malicious, he may recreate the lender pool with all the remaining funds on an address he controls, withdrawing it all in a followup transaction.

Recommendation

I suggest checking the continuing pool output's address in the `validate_eventual_-output_to_pool` function.

Resolution

The issue was fixed by the commit `e20d6c5ddf` .

FTL3-003 Collateral can not be withdrawn

Category	Vulnerable commit	Severity	Status
Logical Issue	`88d07d5a7c`	CRITICAL	RESOLVED

Description

Similar to the issue FTL3-001, the loan token's minting policy does not allow for the token burning. That means that all final branches of loan repayment, those that check that the loan token is burned, are unfeasible. As a result, no collateral can be withdrawn as it unlocks only once the loan is repaid.

Recommendation

I suggest carefully adding in the burning mechanism.

Resolution

The issue was fixed in the commit `3565170f94`.

FTL3-004 Lender can claim the whole collateral in a dutch auction before start

Category	Vulnerable commit	Severity	Status
Design Issue	`610e4c46cd`	CRITICAL	RESOLVED

Description

In a loan with a dutch auction liquidation type, the collateral is auctioned at a premium that is slowly decreasing. If somebody pays for the collateral more than the outstanding debt, it is checked that the rest goes to the borrower — the party being liquidated.

However, there is a `Cancel` mechanism in place which allows the lender to cancel the auction in extraordinary circumstances s.a. the decreasing price dropping too low. Moreover, the lender can cancel the auction before it starts as well. As the start date is always set to the future, the transaction's validity upper bound timestamp, there is some time for him to fit the cancel transaction in. There is no check on the collateral, he can just keep it all. Note that the collateral almost always is of a bigger value than the debt, even when it is being liquidated.

Recommendation

I suggest removing the option of cancelling an auction before it starts. Either way, as can be seen in the issue FTL3-303, the auction actually starts right away.

Resolution

The issue was fixed in the commit `226e399697` . The option to cancel the auction before it starts was removed for dutch auctions started by this contract. It is retained for auctions where no borrower is listed.

FTL3-005 Lender can claim the whole collateral in an auction by malicious address

Category	Vulnerable commit	Severity	Status
Logical Issue	`c49d69ef8c`	CRITICAL	RESOLVED

Description

When a borrower is late with his repayment and the liquidation mode is set to the dutch auction type, the lender can create the auction. In that transaction, he sets his address in the dutch auction's datum. Whoever buys the collateral, he needs to pay the lender his outstanding debt. This address is not checked by any validation.

There are two attacks on this. Both cause the breaking of the validation of the dutch auction when somebody attempts to buy the collateral. The only way that the auction could be resolved, is for the lender to wait until the price decreases to the minimum threshold and then cancel the auction and simply claim the whole collateral. As the collateral is of a higher value than the debt, he has an incentive to do this. The two ways to achieve this:

1. Lender supplies an address where either the payment credential or the stake credential is of the wrong length. Aiken (in contrast to Plutarch) does not protect against this. It is impossible to create a UTxO at an address of an invalid length, so the validation would never see an output to the `ownerAddress`.
2. Lender sets the address' stake credential to a pointer credential. This old way of referencing credentials is not supported in the `correctAmountSentToUser` function and errors out straight away. The function is used only when somebody else buys the collateral; the lender can still wait and cancel the auction. Note: Even if pointer addresses were supported, there is a choice of pointer numbers that would break it down again ☺.

Recommendation

I suggest validating the `ownerAddress` when the dutch auction is being created. Both the payment and the stake credentials need to be of the correct length. I also suggest requiring non-pointer stake credentials.

Resolution

The issue was fixed in the commit `775127a44e` by setting the owner address as the address where the lender token is at the time. Beware, that this issue is not fixed in any way for dutch auctions not originating from this protocol.

FTL3-006 Blocking funds and gaining unfair advantage by adding a programmable token

Category	Vulnerable commit	Severity	Status
Design Issue	`dc3f840801`	CRITICAL	RESOLVED

Description

The scripts support CIP-113 programmable tokens. Any project scripts could be located at the smart wallet payment credential which, among other things, checks that all tokens contained within the UTxO are either not programmable or their programmable logic is run in the transaction. This exposes the protocol to a novel attack vector. The core of the attack is that there will be a new malicious programmable token added to a protocol UTxO at an otherwise normal interaction. Let's explore a few examples:

1. When a lender liquidates a loan by creating a dutch auction, he can add a malicious programmable token to the newly created dutch auction. The token's programmable logic would allow only transactions signed by his key. That means that he can claim the whole collateral for no price as nobody else would be able to buy the collateral from the auction.
2. When a borrower borrows from a pool, she can add a programmable token to the newly created loan UTxO. She can block the UTxO, but that wouldn't help her. However, she can, for example, also allow only actions that are not liquidations of her loan and thus protect her collateral and e.g. be able to keep the loan forever.
3. When borrowing from a pool, a malicious borrower can block the rest of the pool by adding a programmable token that never validates. Alternatively, they can ransom the pool owner by e.g. validating if and only if they send a big portion of the tokens to their controlled address.
4. Newly created repayment can be blocked or ransomed if Ada is the lent asset. Equity UTxOs can be blocked or ransomed as well.

As can be seen in the above examples, with the programmable token support, it is no longer enough to check the DoS protection by limiting the number of tokens. It needs to be checked that no additional programmable token is added to script outputs or outputs belonging to someone else.

Recommendation

I suggest validating that exactly the expected number of tokens is present in every UTxO instead of checking the maximum number of different tokens. This needs to be updated in all the places in the code that check outputs.

Resolution

The issue was partially fixed in the commit `775127a44e` and later fully fixed in the commit `61fb5e21d7`.

FTL3-007 Ada collateral is not protected in requests

Category	Vulnerable commit	Severity	Status
Logical Issue	`c9ad2a4960`	CRITICAL	RESOLVED

Description

Imagine a borrower creating a request and locking their Ada collateral inside. They want to borrow some other token. When a lender comes, they need to pay the principal to the borrower but they can claim the whole collateral for themselves as the `collateralUnchanged` condition in the `validate_output_to_loan` function does not check Ada values at all. As the collateral is bigger than the loan in value, the borrower immediately loses on this.

Recommendation

I suggest checking that at least that amount of Ada is contained within the loan output.

Resolution

The issue was fixed in the commit `12dcddbc44`.

FTL3-008 Lender can disable repaying and liquidate

Category	Vulnerable commit	Severity	Status
Code Issue	`c9ad2a4960`	CRITICAL	RESOLVED

Description

This is a technical vulnerability that exploits the fact that the `hash_output_ref` function errors out if the input's transaction index it tries to hash exceeds 255. All loan repayments use this function to determine the repayment id. As such, if a loan input can not be hashed, it can not be repaid.

To achieve this, let's lend to a request in such a way that the resulting loan output would be on an index > 255. It is enough to create 255 dummy output UTxOs when lending to a request and putting the newly created loan on the 256th position.

A borrower would be unable to repay it; the transactions would start failing. He is not entirely hopeless in all situations, though. It is enough for him to add collateral to his loan, thus setting the new loan UTxO's transaction index to a lower value. He can repay then. However, it is not intuitive to figure this out and he might have get liquidated in the meantime. What's more, it is not always easy or it's straight impossible to get more of some collateral types such as NFTs.

Finally, since the borrower is unable to repay, the lender can liquidate him and take the collateral if the liquidation policy is set right — neither partial liquidations can take place as a repayment is created in those cases. As always, collateral's more valueable than the loan so there's an incentive to do this.

Recommendation

I suggest making sure that no loan is ever put on an index bigger than 255.

Resolution

The issue was fixed by the commit `e20d6c5ddf` . The `hash_output_ref` function was modified to handle the case where the index is bigger than 255 properly.

FTL3-009 Loan token can not be minted for programmable token loans

Category	Vulnerable commit	Severity	Status
Code Issue	`c9ad2a4960`	CRITICAL	RESOLVED

Description

Due to a similar technical issue as described in the issue FTL3-105, the loan token minting policy can not find and thus validate minting of tokens resulting from request and pool inputs that are on the programmable token script credential. It's because it expects the staking credential to reference itself (the loan validator hash) instead of the request and pool validators respectively.

That means that the token can not be created, thus no valid loan can be created for requests or pools containing programmable tokens.

Recommendation

I recommend updating the arguments in the `get_inputs_from_smart_credential` function call in the loan minting policy such that inputs with the programmable token payment credential and request/pool staking credential are found by the call.

Resolution

The issue was fixed in the commit `12dcddbc44`.

FTL3-010 Repayment token can not be minted for programmable token loans

Category	Vulnerable commit	Severity	Status
Code Issue	`c9ad2a4960`	CRITICAL	RESOLVED

Description

Due to a similar technical issue as described in the issues FTL3-105 and FTL3-009, the repayment token minting policy can not find and thus validate minting of tokens resulting from loan inputs that are on the programmable token script credential — e.g. all those whose collateral is programmable. It's because the policy expects the staking credential to reference itself (the repayment validator hash) instead of the loan validator.

That means that the token can not be created, thus no valid repayment can be created. That means that a borrower might not have the opportunity to repay his loan and get liquidated.

Recommendation

I recommend updating the arguments in the `get_inputs_from_smart_credential` function call in the repayment minting policy such that loan inputs with the programmable token payment credential and loan staking credential are found by the call.

Resolution

The issue was fixed in the commit `12dcddbc44`.

FTL3-011 Protocol is unfeasible due to usage of `get_outputs_to_smart_credential`

Category	Vulnerable commit	Severity	Status
Code Issue	`c9ad2a4960`	CRITICAL	RESOLVED

Description

This issue is about a similar technical issue as described in the issues FTL3-105, FTL3-009 and FTL3-010. The function `get_outputs_to_smart_credential` is used to find an output on either a traditional payment credential or a programmable token credential with a particular owner referred by the staking credential. The latter case does not work properly in some cases as the staking credential that is looked for is supplied incorrectly.

There are three cases of incorrect parameters supplied to this function resulting in the unfeasibility of the protocol in each case:

- When a lender lends to a request, the newly created loan output can be created on the programmable credential with the loan staking credential. However, the validator looks for the request staking credential and thus can't find the loan output. However, no loan token ever gets to a non-loan payment credential and so requests can not be lent to.
- Analogous to the previous point, when it is borrowed from a pool, the loan output's not looked for correctly. That means that it can not be borrowed from a pool.
- When a loan is being liquidated and a dutch auction is created, the dutch auction output is not looked for correctly. The staking credential is expected to be that of the loan credential. This means that the collateral can not go to the auction. It can be forever locked inside an unspendable, invalid loan output.

All of the above examples apply only if there are programmable tokens involved.

Recommendation

I recommend updating the arguments in the `get_outputs_to_smart_credential` function call in the mentioned cases, focusing on the `withdrawScriptCredential` argument.

Resolution

The issue was fixed in the commit `12dcddbc44`.

FTL3-012 Dutch auction's borrower compensation goes to the lender

Category	Vulnerable commit	Severity	Status
Code Issue	`c9ad2a4960`	CRITICAL	RESOLVED

Description

In case the dutch auction is successful and the raised value exceeds the remaining debt, the rest is paid to the borrower address specified in the dutch auction's datum. However, the value is not set correctly in the loan validator. It is set to the lender address instead as the `lenderBondInput` is supplied to the `validate_output_to_dutch_auction` function's `borrowerBondInput` parameter. As a result, the borrower always loses the whole collateral's worth of value.

Recommendation

I recommend correcting the variable supplied to the `borrowerBondInput` parameter to refer to the actual borrower bond input. A reference input is probably more suitable here. Also, I'd recommend highlighting to the users that they might receive liquidation compensation on the address where they hold the bond — to make sure that their compensation is not sent to an unspendable script address.

Resolution

The issue was fixed in the commit `775127a44e` .

FTL3-013 AMM formulas are incorrect

Category	Vulnerable commit	Severity	Status
Logical Issue	`c9ad2a4960`	CRITICAL	RESOLVED

Description

One of the oracle types is an oracle that returns the amount of tokens A and tokens B inside a constant-product AMM liquidity pool. Based on this, it is estimated what the price of converting one into the other would be, taking into account protocol fees as well as slippage. However, these formulas are wrong. As data from oracles as well as their interpretation are insanely crucial for the protocol, everything can go wrong with this.

Recommendation

As the simplest suggestion, I suggest re-doing the way AMM logic functions are written; writing them in a similar way to how the DEXes verify the swap values, ideally sticking as close to the same DEX code as possible.

They also do not compute all the variables on-chain. Sometimes, it is easier to supply additional information via redeemers instead of computing multiple inequalities as would be the case in the `token_b_needed_to_purchase_token_a_in_AMM_pool` function. Say that the `token_b_needed` is known and supplied via a redeemer and then it is just verified. Inequalities turned into simple formulas.

Furthermore, I'd recommend choosing a DEX that is planned to be sourced for values and naming the variables exactly the same. It helps to avoid mistakes. Additionally, I'd suggest covering this logic with numerous tests that directly test the logic against the logic from the chosen DEX. If numerous are planned, test against them all. As an example, you can refer to this WingRiders computation[1]. Note the rounding there as well.

Finally, I suggest sanitizing the function arguments, including but not limited to checking that the numbers are positive, the pool has enough reserves for the swap; and in case of using any inequalities, checking that they were not multiplied by a negative number (requiring the inequality sign change).

[1] https://github.com/WingRiders/dex-v2-contracts/blob/master/src/DEX/Pool/ConstantProduct.hs#L365

Resolution

The issue was fixed by the commit `4b1ee3fe68`. The `Pooled` oracle data type was phased out and the mentioned functions were removed as a result. DEX prices can be still sourced off-chain and feeded via an `Aggregated` oracle type. I'd like to stress that if that is done, the computation needs to be done off-chain. The on-chain trusts it.

The `Pooled` oracle type is still present in the codebase; however, the validation fails if used.

FTL3-014 Wrong arguments in conversion from Ada to token

Category	Vulnerable commit	Severity	Status
Logical Issue	`c9ad2a4960`	CRITICAL	RESOLVED

Description

The `get_lovelace_amount_in_token_currency` function converts a fixed lovelace amount into the token currency. For that, it uses the `token_b_needed_to_purchase_token_a_in_AMM_pool` function which is supposed to compute the token B required to purchase `wanted_token_a_amount` of token A according to its description. Wanted token A amount is set to the fixed lovelace amount. However, as mentioned in the `Pooled` oracle data type, lovelace is always the token B. Hence, it makes a mistake of supplying B amounts to A amounts and vice versa. That results in invalid numbers in protocol foundations.

Recommendation

As this is the only usage of the function, I suggest re-labelling token A to token B and vice versa inside the `token_b_needed_to_purchase_token_a_in_AMM_pool` function, including in its name.

Additionally, I'd add a comment to the function explaining that it charges fees related to the conversion, if any, to the opposite conversion. This can be explained better by naming as well, e.g. by naming the function similar to `how_much_tokens_convert_to_fixed_lovelace`. Its call makes sense only in hypothetical scenarios such as determining the collateral value in such a way that the collateral value converted would yield at least this amount of lovelace, etc. It is not trivial to see this from the function alone and it is important to clarify.

Resolution

The issue was fixed by the commit `4b1ee3fe68`. The `Pooled` oracle data type was phased out and the mentioned functions were removed as a result.

FTL3-015 Healthy loans can be liquidated

Category	Vulnerable commit	Severity	Status
Logical Issue	`c9ad2a4960`	CRITICAL	RESOLVED

Description

This issue is about liquidations using loan-to-value (LTV) comparisons. Loan to value number in overcollateralized loans is a number between 0 and 1. The bigger the number, the bigger the loan compared to the collateral value and thus the more dangerous or unhealthy the loan is. The liquidation LTV threshold is noted in the loan datum.

The `can_liquidate` function compares the current LTV against the liquidation threshold and decides whether the loan can be liquidated or not. However, the logic is reversed. It says that liquidation can happen if the liquidation threshold is greater than the current LTV. In other words, if the threshold is more dangerous than the current ratio. It allows liquidation for healthy loans only.

Recommendation

I suggest reversing the logic in the `can_liquidate` function, so that liquidations are allowed for loans where the current LTV is greater than or equal to the liquidation threshold.

Resolution

The issue was fixed in the commit `12dcddbc44`. Loans can now be liquidated only when the current LTV is greater than the liquidation threshold. Equality is not enough, though.

FTL3-016 Amortization formula is wrong

Category	Vulnerable commit	Severity	Status
Logical Issue	`c9ad2a4960`	CRITICAL	RESOLVED

Description

When a loan is taken with the `InterestOnRemainingPrincipal` repayment method, the amortization formula is used to determine the installment amounts. However, it is incorrect in the code. Instead of:

$$\frac{principal \times i \times (1+i)^n}{(1+i)^n - 1}$$

it uses

$$\frac{(principal \times i \times (1+i))^n}{(1+i)^n - 1}$$

. The latter is significantly bigger and would result in massive overpaying of the loan or, more likely, a default and a liquidation of the whole collateral.

Recommendation

I suggest fixing the formula as described.

Resolution

The issue was fixed in the commit `12dcddbc44` .

FTL3-017 Recasting does not work well with the amortization formula

Category	Vulnerable commit	Severity	Status
Design Issue	`c9ad2a4960`	CRITICAL	RESOLVED

Description

A recast can happen any time during a loan and any amount can be recast. The recast amount is directly subtracted from the initial principal. Based on this new principal, a new fixed installment is computed; not changing the rate and repaid/due installment amounts.

However, that ignores the fact that if this recast happens mid-term, a portion of the initial principal has already been repaid in those repayments.

Furthermore, the term is not updated. If the recast happens after the 3rd out of 12 installments, the formulae expect that the new outstanding principal, however computed, should be repaid after 12 installments. However, that's not true. Only 9 installments are pending.

Recommendation

I suggest modifying the recasting logic of the `InterestOnRemainingPrincipal` repayment method. Each repayment includes both a part of principal and interest. When a recast happens mid-term, it is important to update both the term and the new principal. The new principal is the outstanding principal minus the recast amount. The outstanding principal after k repayments can be computed iteratively or there's a known closed formula as well. The term needs to be updated such that the per-installment rate remains the same, but the total number of installments used in the fixed payment amortization formula n is reduced to the number of outstanding repayments.

Resolution

The issue was fixed by the commit `e4f6501c15` .

FTL3-018 Recasting on due loan installments avoids interest and penalties

Category	Vulnerable commit	Severity	Status
Design Issue	`c9ad2a4960`	CRITICAL	RESOLVED

Description

A recast can happen any time during a loan and any amount can be recast. It does not check whether installments that are already due are repaid, though. That means, taken to the extreme, that the principal can even be fully repaid only in the end of the loan with no installments paid up to that point; and the loan could be closed as fully repaid that way. Technically, this is possible as computing the remaining debt after the recast while supplying 0 principal yields 0.

Recommendation

I suggest ensuring that all due installments and one more are repaid before allowing a recast. I suggest one more simply such that the term is not changed in the end of an installment period when there should be a bigger pending interest accumulated up to that point.

Resolution

The issue was fixed by the commit `8d0e95f6bc` .

FTL3-019 Inconsistent perpetual loan interest computation

Category	Vulnerable commit	Severity	Status
Logical Issue	`c9ad2a4960`	CRITICAL	RESOLVED

Description

There are two different calculations for the perpetual loan's current interest rate. In `get_remaining_debt`, the linear increase is multiplied by the base rate (i.e. $rate + (rate \times increase)$), whereas in `get_installment_amount`, the increase is simply added to the rate (i.e. $rate + increase$). This leads to mismatched interest values and thus total values in two critical functions that need to match.

Recommendation

I suggest using one consistent interpretation of the rate increase. If the design intends a percentage-based scaling, apply $rate + (rate \times increase)$ everywhere. If it's a flat additive factor, use $rate + increase$ throughout.

Resolution

The issue was fixed in the commit `3bb4c59d7f` .

FTL3-020 Remaining debt on perpetual loans does not assume previous payments

Category	Vulnerable commit	Severity	Status
Logical Issue	`c9ad2a4960`	CRITICAL	RESOLVED

Description

In the perpetual loan scenario, the `get_remaining_debt` function always calculates the total debt, principal plus total interest, as if no interest payments were ever made. It simply uses the full timespan from the lend date; ignoring that periodic installments may have already covered some or all of the accrued interest. As a result, borrowers could be double-charged for interest they've already paid.

Recommendation

I suggest accruing interest only from the last repayment forward. This ensures the final payoff amount truly reflects unpaid interest rather than recalculating from the initial lend date each time.

Resolution

The issue was fixed by the commit `e20d6c5ddf`. The perpetual loan's logic has changed a lot since the issue was reported. But the calculation of the remaining debt takes into account the previous payments — even though the logic itself is not 100% correct at this commit yet.

FTL3-021 Dutch auction payments can break due to checking bond addresses

Category	Vulnerable commit	Severity	Status
Logical Issue	`aef2d5dd0a`	CRITICAL	RESOLVED

Description

After some refactors, the loan validator was updated to take the address of the borrower bond output as the borrower address, and the address where the lender bond is as the lender address. Those are then used in the configuration of the dutch auction. There, they are used in the `correctAmountSentToUser` validation function.

This issue is about what can go wrong with this approach inside the function:

- First, as some other issues mentioned, it validates if and only if an address got from that contains an inline staking credential. If the lender does not keep his bond at an address with such a staking credential, s.a. if he has no staking credential there or uses a pointer staking credential, he can DoS the auction and keep the whole liquidation even if partial liquidations are enabled.
- Then, the validation can validate if the collateral is sent either to the address specified or to the programmable account credential with the user staking credential. In the latter case, there can be an unspendable programmable token attached by a malicious party, effectively blocking the follow-up spend of the UTxO.
- Finally, a user probably doesn't hold the bond on the programming tokens' credential. That means that the staking credential of that address is likely just a staking credential. Allowing a repayment address of the programmable credential and that staking credential might not be handled well by even the wallets supporting the CIP-113.

Recommendation

I recommend updating the dutch auction's `correctAmountSentToUser` to:

- Distinguish between a `userAddress` that is on the programmable credential and the normal pubkey address.
- Use the expected programmable credential's staking key (payment key in case of normal user address).

- Do not allow the lender to DoS the process by making the validation fail on a non-present or a pointer staking credential.
- Similar to the issue FTL3-006, tightly control the amount of tokens in the output — especially in the programmable collateral scenario.

Furthermore, explain to users that the address where they hold their bonds can be used in the way mentioned here. Among other things, they should not put the bonds on a script payment credential because they might not be able to unlock the collateral if it is a programmable token and the script is used as the staking credential. Also, the address needs to be able to receive tokens.

Resolution

The issue was fixed by the commit `4ff2165210`.

FTL3-022 Perpetual loan recasting logic is incorrect

Category	Vulnerable commit	Severity	Status
Design Issue	`aef2d5dd0a`	CRITICAL	RESOLVED

Description

The perpetual loan recasting logic has several flaws that lead to incorrect calculations and potential loss of funds:

- It reduces the principal amount immediately by subtracting the `principalPaid` from `datum.principalAmount` without first ensuring all accrued interest has been paid.
- The remaining debt calculation is based on the reduced principal, which means interest accrued on the full principal amount up to that point may not be fully accounted for.
- The `is_recasting_permitted` check only verifies that there is no due installment, but that doesn't guarantee all accrued interest has been settled as the installments do not cover the whole interest amounts.

Recommendation

I suggest restructuring the recasting logic to:

- First calculate the total current debt including all accrued interest based on the original principal amount.
- Verify that the `principalPaid` amount covers both the whole unpaid accrued interest up to the current time and the desired principal reduction amount.
- Only after confirming all interest is paid, allow the principal reduction by the appropriate principal portion of the recast payment.

Resolution

The issue was fixed by the commit `c7b9d1cd2d` .

FTL3-023 Loan inputs with programmable assets bypass action validator checks

Category	Vulnerable commit	Severity	Status
Code Issue	`db29a2fe1d`	CRITICAL	RESOLVED

Description

All four loan action validators incorrectly filter loan inputs when using programmable assets. In all these validators, the `get_inputs_from_smart_credential` function is called with the wrong withdraw script credential set to the action validator's credential, but it has to be set to the loan script credential in order to find loan inputs on programmable credentials.

This causes loan UTxOs containing programmable collateral to be missed by the action validators, allowing them to be spent without any validation. Attackers can freely unlock loan UTxOs with programmable collateral, bypassing liquidation conditions, repayment requirements, and other loan constraints.

Recommendation

Change the `credential` variable to loan script credential inside the `get_inputs_from_smart_credential` calls in all four loan action withdraw validators.

Resolution

The issue was fixed by the commit `4ff2165210`.

FTL3-024 Wrong action credential allows borrowers to unlock programmable collateral

Category	Vulnerable commit	Severity	Status
Code Issue	`db29a2fe1d`	CRITICAL	RESOLVED

Description

Three of the four loan action validators (`loan_change_collateral_action.ak`, `loan_recast_action.ak`, `loan_repay_action.ak`) incorrectly filter ongoing loan outputs when using programmable assets. In these validators, the `get_outputs_to_smart_credential` function is called with the wrong withdraw script credential.

The `credential` parameter is the action validator's credential, but it should be set to the loan credential. This allows borrowers to send continuing loan outputs to addresses controlled by the action validator's credential instead of the proper loan spend script credential. This in turn, after fixing the previous issue FTL3-023, makes them invisible to the validator and so they become easily withdrawable since they're not properly locked to the loan spend script. That effectively allows borrowers to unlock their collateral without any constraints.

Recommendation

Change the `credential` parameter passed to `check_repay` and similar functions to use the loan script credential.

Resolution

The issue was fixed by the commit `4ff2165210`.

FTL3-025 Wrong receipt condition allows blocking funds with programmable assets

Category	Vulnerable commit	Severity	Status
Logical Issue	`db29a2fe1d`	CRITICAL	RESOLVED

Description

The receipt token validation logic is implemented incorrectly. The number of tokens allowed inside a UTxO is decided based on a `receiptCondition` logic which is set as follows:

```
let receiptCondition =
    repaymentReceipts == False || quantity_of(
      equityOutput.value,
      repaymentPolicyId,
      utils.hash_output_ref(loanInputOutputReference),
    ) == 1
```

When `repaymentReceipts == False` (repayment receipt token not required), the condition is always `True` regardless of whether the receipt token is present in the output. Based on that, the code potentially allows one more arbitrary token to be added there.

That allows attackers to add a programmable token there that never validates, effectively blocking the UTxO. Both equity and repayment outputs can be blocked or ransomed this way.

Recommendation

Fix the `receiptAssetCount` logic to allow the correct receipt token, but still prevent other tokens from being added to the output.

Resolution

The issue was fixed by the commit `4ff2165210`.

FTL3-026 Programmable collateral sent to uncontrollable auction credential is lost

Category	Vulnerable commit	Severity	Status
Logical Issue	db29a2fe1d	CRITICAL	RESOLVED

Description

The `is_output_to_smart_credential` function was incorrectly updated when fixing another issue. It now doesn't properly support programmable assets controlled by scripts. It is used once that way in the codebase, when creating a dutch auction output.

```
pub fn is_output_to_smart_credential(
  output: Output,
  spendCredential: Credential,
  stakeCredential: Option<StakeCredential>,
  smartTokensSpendScriptHash: ByteArray,
) {
  or {
    //Native tokens
    and {
      output.address.payment_credential == spendCredential,
      output.address.stake_credential == stakeCredential,
    },
    //Smart tokens or native tokens accidentally sent to the smart account
    and {
      output.address.payment_credential == Script(smartTokensSpendScriptHash),
      output.address.stake_credential == Some(Inline(spendCredential)),
    },
  }
}
```

For programmable assets, this function forces the stake credential to be `Some(Inline(spendCredential))`. When used in dutch auction creation with `dutchAuc-`

`tionSpendScriptHash` as the `spendCredential`, programmable collateral gets sent to the programmable payment credential and this staking credential. However, `dutchAuctionSpendScriptHash` is a `general_spend` script that is unable to control programmable tokens' transfer, making these funds — auctioned collateral — permanently lost.

This function works for user wallets and that's why it was updated, but fails for script-controlled addresses where the spend credential and the credential able to control programmable token transfers are different from each other.

Recommendation

Revert the function's definition to properly handle script-controlled programmable assets as well, ensuring the stake credential can actually control the programmable tokens when scripts are involved.

Resolution

The issue was fixed by the commit `4ff2165210`.

FTL3-027 Zero liquidation penalty incorrectly skips equity return to borrower

Category	Vulnerable commit	Severity	Status
Logical Issue	`db29a2fe1d`	CRITICAL	RESOLVED

Description

A recent change modified a partial liquidation condition in the `loan_claim_action.ak` file on line #247 from `< 0` to `<= 0`, causing equity to be wrongfully withheld from borrowers in case the liquidation discount, the `partialLiquidationPenaltyPerMille` variable, is set to exactly zero meaning no liquidation discount.

If that happens, the condition becomes `True` and no equity output is required to be created for the borrower upon liquidation. However, even with zero liquidation penalty, if there is positive equity in partial liquidation scenario (collateral value exceeds remaining debt), the borrower should still receive that equity back.

Recommendation

Revert the condition back to the strict inequality in the `loan_claim_action.ak` file:

```
or {
  partialLiquidationPenaltyPerMille < 0,  // Revert to < 0
  equity <= 0,
  {
    // ... equity creation logic
  }
}
```

Resolution

The issue was fixed by the commit `4ff2165210`.

FTL3-028 Malicious parties can block transactions by holding bonds without stake credentials

Category	Vulnerable commit	Severity	Status
Design Issue	`db29a2fe1d`	CRITICAL	RESOLVED

Description

Multiple validators now expect bond addresses to have inline stake credentials, creating denial-of-service attack vectors where malicious parties can block critical loan operations simply by placing their bond on an address with no staking credential, with a pointer staking credential, etc.

```
expect Some(Inline(lenderBondStakeCredential)) =
    lenderBondAddress.stake_credential
```

Attack Vectors:

- **Lender blocks repayments** — Lender can block repayments as the check is present in the `check_repay` function. By making repayment transactions fail, he forces the loan to become late and enables liquidation.
- **Lender blocks recasts** — Same mechanism prevents borrowers from recasting loans (`loan_recast_action.ak` line #99).
- **Borrower blocks liquidations** — Borrowers can prevent liquidations by putting bonds on addresses without inline stake credentials (`loan_claim_action.ak` line #226).

Recommendation

Never enforce something about the type of an address of a different party. Replace the logic such that it validates gracefully for any type of address the bond is located on.

Resolution

The issue was fixed by the commit `c23560b30f` .

FTL3-029 Bond address trusted without bond presence verification

Category	Vulnerable commit	Severity	Status
Logical Issue	`db29a2fe1d`	CRITICAL	RESOLVED

Description

There was a refactor whereas some outputs could be created directly at a user's address. To determine the address, a reference input is included in the transaction that contains the party's bond token. However, the bond addresses are actually extracted from reference inputs without verifying that the corresponding bond NFTs are actually present in those reference inputs! This allows attackers to redirect payments to addresses they control.

Attack Scenarios:

- **Borrower steals repayments** — Borrower points `lenderBondRefInputIndex` to a reference input with their own address, receiving all loan repayments instead of the lender. The presence of lender bond token in that reference input is not verified.
- **Borrower steals recasts** — Same mechanism allows borrowers to redirect recast principal payments to themselves.
- **Lender steals equity** — Lender points `borrowerBondRefInputIndex` to their own address, receiving borrower equity from partial liquidations.

The validators only verify bond NFT presence in outputs, not in the reference inputs used for address extraction.

Recommendation

Add the necessary bond NFT presence check whenever a bond input is referenced to extract an address to send some value to.

Resolution

The issue was fixed by the commit `4ff2165210`.

FTL3-030 Datums can not be parsed

Category	Vulnerable commit	Severity	Status
Logical Issue	`10a0a94ee8`	CRITICAL	RESOLVED

Description

Recent changes to the code introduced a different way of parsing datums for performance reasons, using `builtin.un_list_data` on the datum directly. However, if the datum type is not changed to list only, but it still conforms to a datum type structure s.a. `ConfigDatum`, then that can not be parsed as a list. This has been demonstrated by an aiken test.

The implications of this are protocol-wide unfeasibility. For example, the config validators make sure the config datum conforms to the mentioned `ConfigDatum`, but other validators try to parse it as the list, which always fails.

Recommendation

Make sure the feasibility of the data parsing is not impacted — there is a constructor at the top level, so either `un_constr_data` or `unconstr_fields` usage is needed. As there is a big loss of type safety with the taken approach, try to abstract the parsing/unparsing functionality into a single file, avoiding copy-paste indexing-like errors across the validators and covering this file with many tests, so changes s.a. moving a variable's position inside a data type are easily caught by the tests.

Resolution

The issue was fixed by the commit `c23560b30f` .

FTL3-031 Wrong config index extracts incorrect loan policy id

Category	Vulnerable commit	Severity	Status
Logical Issue	`c23560b30f`	CRITICAL	RESOLVED

Description

All loan validators extracted `loanPolicyId` from the wrong config index, using index 2 instead of the correct index 6:

```
let loanPolicyId = builtin.un_b_data(utils.safe_list_at(config, 2))
```

This affects 7 validators: all 4 loan action validators, as well as the `pool.ak`, `request.ak`, and `repayment.ak` validators. Instead of retrieving the actual loan policy id, the code retrieves the pool policy id.

As a result, if loan UTxOs contain programmable tokens, the loan action validators do not find proper loan inputs (they are looking for pool inputs instead) and so they do not perform any validation, allowing for full value extraction by anyone.

Recommendation

Update all instances to use the correct config index 6 for `loanPolicyId` extraction from the config datum.

Resolution

The issue was fixed by the commit `e4f6501c15`.

FTL3-032 LTV is calculated based on the initial principal

Category	Vulnerable commit	Severity	Status
Design Issue	`c7b9d1cd2d`	CRITICAL	RESOLVED

Description

The `can_liquidate` function computes the loan-to-value (LTV) ratio using only the initial principal and the collateral value. This approach does not account for accumulated interest over time or any principal that has already been repaid, and thus bases the LTV solely on the original principal versus the current collateral value.

This logic is used to determine eligibility for liquidation and to decide whether collateral can be reduced to a desired amount. As a result, discrepancies can lead to undercollateralized loans that cannot be liquidated, or to premature liquidations if the principal is almost repaid.

Recommendation

Use the entire remaining debt (including interest and minus any repaid principal) as the "loan" part of the LTV calculation in the `can_liquidate` function.

Resolution

The issue was fixed by the commit `6c94d00585` .

FTL3-033 Repayment increments wrong field causing eventual collateral loss

Category	Vulnerable commit	Severity	Status
Code Issue	`c7b9d1cd2d`	CRITICAL	RESOLVED

Description

The `validate_eventual_output_to_loan_for_repayment` function incorrectly updates the `LoanDatum` after repayment by incrementing `doneRecasts` (field 0) instead of `repaidInstallments` (field 3). This happens because the code uses positional field updates but `repaidInstallments` is not the first field in the `LoanDatum` structure anymore — the bug was introduced when the fields' order was updated and this function was not thought of.

As a result, the loan appears unpaid even after successful payments, allowing lenders to claim collateral from borrowers who might have actually repaid their installments.

Recommendation

Update the correct `LoanDatum` field to correctly increment `repaidInstallments` instead of `doneRecasts` in the repayment validation function.

Resolution

The issue was fixed in the commit `b5bcab310f` .

FTL3-101 DEX oracle computation uses hard-coded fees

Category	Vulnerable commit	Severity	Status
Design Issue	`0592e08738`	MAJOR	RESOLVED

Description

There is an option to use an oracle containing the amounts of a liquidity pool's assets from an AMM decentralized exchange source. To determine the price of an asset, the constant product formula is used. This tries to count-in the impact the sell would have on the price. Fees are set at 0.3% approximating what DEXes on Cardano currently use. However, it is hardcoded at this value. This means that for any DEX or pool with different fee structure s.a. those that incur project fees, as well as any non-constant product pools such as stableswap pools, this can not be used or would yield incorrect results.

Recommendation

I suggest putting the pool fees in the `Pooled` subtype of the `OraclePriceFeed` data type. As the feed can come from different pools from different sources, this gives the protocol the flexibility to update the computation without upgrading the protocol. If stableswap pools are to be supported, the stableswap formula needs to be implemented as well.

Resolution

The issue was fixed in the commit `226e399697` . The pool fee in per mille can be set in the `Pooled` subtype of the `OraclePriceFeed` data type. No stableswap support has been added in this commit — Ada stableswap pools s.a. OADA will use `Dedicated` oracle type and stable-to-stable pools will keep being calculated via their Ada price comparison.

FTL3-102 Ada in expired requests is vulnerable to double satisfaction

Category	Vulnerable commit	Severity	Status
Logical Issue	`dc3f840801`	MAJOR	RESOLVED

Description

If the same borrower has multiple expired requests, part of his Ada can be stolen. As Ada can be a collateral asset as well, it might be a considerable amount that is locked there. Anybody can cancel expired requests and take a specified penalty fee in Ada out of the request. It is checked that the rest of the request value goes to the borrower. For simplicity, an index is provided in the redeemer and the output on that index is checked to contain the borrower's compensation.

It is not checked that the same output is not referenced twice in the same transaction, though. If it is, only the bigger compensation needs to be created. The rest can be freely taken.

Note that the requests need to be equal in the other tokens; Ada collateral requests are the most probable victims.

Recommendation

I suggest checking that the same `borrowerOutputIndex` is not used multiple times.

Resolution

The issue was fixed in the commit `3bb4c59d7f` by checking an output at a deterministic index instead of using the index provided in the redeemer.

FTL3-103 Too big loan can liquidate the borrower

Category	Vulnerable commit	Severity	Status
Design Issue	`c9ad2a4960`	MAJOR	RESOLVED

Description

Let's assume a borrower who creates a request and locks collateral inside. He wants to borrow and achieve a target LTV there. He is okay with liquidations taking place if his collateral loses value too much, it attracts more lenders.

The contracts protect the borrower in a way that no lender can lend him too little and still lock the whole collateral. However, they do not enforce a maximum loan that a lender can lend him.

Imagine a borrower lending too much. The borrower would still need to pay interest on the whole amount. He might not have wanted to borrow that much. Furthermore, there's a bigger threat. If oracle-based liquidations are enabled and partial liquidation disabled, the lender might have pushed the real LTV of the loan over the liquidation threshold straight from the beginning. He might claim the whole collateral that way.

Recommendation

After implementing the `minPrincipal` variable semantics from the issue FTL3-306, I recommend adding a `maxPrincipal` value as well. That way, the borrower can choose what the biggest loan he wants to take is; either in terms of the amount of tokens or its value compared to the collateral he locked in.

Resolution

The issue was fixed by the commit `4b1ee3fe68`. The `maxPrincipal` variable was added and enforced in the request script — note that it is represented as an exact number only, not a dynamic LTV-based value.

FTL3-104 Cross-script double satisfaction

Category	Vulnerable commit	Severity	Status
Design Issue	`c9ad2a4960`	MAJOR	RESOLVED

Description

There are multiple workflows where it is checked that a certain party is being paid a certain amount. These cases are generally vulnerable to double satisfaction if other script inputs are (could be) present. Those script inputs do not have to be related to this protocol at all. It is enough that they verify that the party is being paid as well and those two validations are not mutually exclusive. It is a mutual responsibility to try to eliminate these attack vectors. If no protocol did this, the attacks would be real and the consequences would not necessarily be small.

As a special case of this on just the scripts of this protocol, if the same borrower creates a request and also is part of a dutch auction in the same asset, a payment UTxO can be created for him that would satisfy both the dutch auction script's payment expectations and the request script's principal payment expectations while paying him just the bigger of the amounts.

Recommendation

I suggest checking that no additional script input is present in certain transactions that expect a direct payment, such as when lending to or cancelling a request (principal payment and borrower compensation are exposed) and when interacting with a dutch auction (payments to the owner and borrower are both exposed).

Resolution

The issue was fixed by the commit `38b8354e2a`. Direct payments are now checked to contain an additional datum mentioning the output reference of the input they are coming from making the cross-script double satisfaction attack impossible.

FTL3-105 Permissioned conditions not enforced for programmable tokens

Category	Vulnerable commit	Severity	Status
Code Issue	c9ad2a4960	MAJOR	RESOLVED

Description

Requests and pools might request an additional validation of allowing only specific lenders to lend to them or requiring a kyc'd party. Let's focus on requests and the request specific lenders condition for simplicity, but the problem is analogous for pools and pool kyc token additional condition.

Request UTxOs are on two possible addresses, either on a general spend script referring to the request validation or they contain programmable tokens and are on the programmable token script hash referring the request validation in the staking credential.

The validation of the request specific lenders condition looks for all the requests on either the general spend script (correct) or on the programmable token script referencing the request specific lender key on the staking part (wrong). Thus, it doesn't find and doesn't check request inputs that are on the programmable token payment credential with the request hash staking credential.

Recommendation

I recommend changing the following code that looks for request inputs in the request specific lenders validator from this:

```
get_inputs_from_smart_credential(
  self.inputs,
  Script(requestSpendScriptHash),
  credential,
  smartTokensSpendScriptHash,
)
```

to:

```
get_inputs_from_smart_credential(
  self.inputs,
  Script(requestSpendScriptHash),
```

```
  requestPolicyId,
  smartTokensSpendScriptHash,
)
```

Furthermore, an analogous change is required in the pool kyc token's validation.

Resolution

The issue was fixed in the commit `12dcddbc44`.

FTL3-106 Time unit change error disables re-casts

Category	Vulnerable commit	Severity	Status
Code Issue	`4b1ee3fe68`	MAJOR	RESOLVED

Description

There was a recent change where the `installmentPeriod` was updated to be in hours as opposed to milliseconds. This new semantics was not properly reflected in all places it was used, though. In the `get_due_installments` function, the result of the function divides `timePassed` in milliseconds by the `installmentPeriod` which is now in hours. That results in seemingly very big number of due installments. The function is called to determine eligibility for recasts, almost always disabling it as it looks like there are many due installments.

Recommendation

I suggest converting the `installmentPeriod` variable to milliseconds before the `timePassed` variable is divided by the number in the `get_due_installments` function. I also discourage non-essential refactors and changes of semantics at this point.

Resolution

The issue was fixed in the commit `8d0e95f6bc` .

FTL3-201 Minting multiple repayment tokens is nearly unfeasible

Category	Vulnerable commit	Severity	Status
Logical Issue	`c49d69ef8c`	MEDIUM	RESOLVED

Description

The repayment minting policy supports the minting of multiple repayment tokens in a single transaction. That is useful in a case when multiple loans' installments are repaid in a single transaction. However, the way the minting policy verifies that no additional tokens are minted is near impossible to satisfy for 2+ repayment tokens.

It requires an exact match of expected and actual token mints. The actuals are all the positive repayment token mints from the transaction's mint value. The mint value is a dictionary and this list is ordered by the asset name. It compares it with a constructed list of the expected mints. This is constructed by taking the loan inputs, considering just a subset of them as not all actions require a repayment token mint, hashing the input's reference to get the token name and concatenating these records.

Let's focus on the ordering. Minted token names are lexicographically ordered. They need to be equal to the hashes of some loan inputs from that transaction. The inputs' order is also lexicographically sorted by their output reference. However, their hash is unpredictable and, more often than not, does not represent the same order. As a result, minting multiple repayment tokens is almost always unfeasible for 3+ tokens, there's a 50% chance for two tokens and it works fine for a single token.

Recommendation

I suggest sorting the `expectedMintedNFTs` before comparing them to the `mintedNFTs` in the repayment minting policy.

Resolution

The issue was fixed in the commit `d7d64ae28f` .

FTL3-202 Request can not be cancelled after expiration by a different party

Category	Vulnerable commit	Severity	Status
Logical Issue	`dc3f840801`	MEDIUM	RESOLVED

Description

Even though there is a specific workflow that allows any party to cancel an expired request, the workflow is not feasible. It is checked that the whole request value minus the penalty fee goes to the borrower. However, it is forgotten that a request contains a request token which is burned in that transaction. As a result, it can not be part of the borrower's compensation output.

Recommendation

I suggest amending the `validate_collateral_less_penality_output` function in a way that does not expect the request token to be in the borrower's output. Also, I'd suggest correcting the typo in the function name, changing "penality" into "penalty".

Resolution

The issue was fixed by the commit `8d0e95f6bc` .

FTL3-203 It is possible to lend to an expired request

Category	Vulnerable commit	Severity	Status
Logical Issue	`c9ad2a4960`	MEDIUM	RESOLVED

Description

A borrow request has an expiration date after which anybody can cancel it and claim a penalty fee. However, it is still possible to lend to an expired request as there is no check checking whether or not the request expired.

Recommendation

I suggest adding a check allowing lending to only active requests.

Resolution

The issue was fixed by the commit `8d0e95f6bc` .

FTL3-204 Pool might be blocked until recreated

Category	Vulnerable commit	Severity	Status
Code Issue	`c9ad2a4960`	MEDIUM	RESOLVED

Description

An attacker can block the liquidity inside a pool by exploiting the same vulnerability as in FTL3-008, the `hash_output_ref` function's inability to hash indices exceeding 255. He can borrow the smallest amount from the pool and recreate the pool on too big an index by adding dummy UTxOs in between. By doing this, nobody is able to borrow from the pool anymore as the loan id is determined by hashing the pool input's output reference.

The pool owner can cancel the pool and recreate it on a small index. However, especially for pools owned by companies, the recreation process might be long and impractical, and the liquidity blocked substantial. To sum up, the attacker might block a lot of liquidity this way, even though not permanently.

Recommendation

I suggest checking that the pool is not recreated on an index that is too big. Alternatively, you might adjust the `hash_output_ref` function to allow for bigger indices.

Resolution

The issue was fixed by the commit `e20d6c5ddf` . The `hash_output_ref` function was modified to handle the case where the index is bigger than 255 properly.

FTL3-205 Too small Ada equity makes liquidation impossible

Category	Vulnerable commit	Severity	Status
Logical Issue	`c9ad2a4960`	MEDIUM	RESOLVED

Description

In case of partial liquidations with Ada principal, leftover Ada is sent to the borrower. However, it is checked that exactly the computed equity amount is sent to the borrower. Since there are min Ada requirements on any UTxO, it might be impossible to create such a UTxO in cases when the LTV is close to 1 and thus the liquidation would be unfeasible altogether.

Recommendation

I suggest checking that at least the amount is present in the equity compensation output instead of checking exactly the amount computed.

Resolution

The issue was fixed in the commit `4fda819d4b`.

FTL3-206 It might be impossible to add collateral to a non-specific asset collateral loan

Category	Vulnerable commit	Severity	Status
Logical Issue	`c9ad2a4960`	MEDIUM	RESOLVED

Description

It is possible to supply a collateral using a method that values all assets under a specified policy id as equal. Furthermore, naturally, it should be possible to add collateral to those loans as well and the code allows that. The comparison of the value in the input and output loan UTxOs compares Ada value strictly, meaning Ada can not be increased in the output UTxO unless that's the collateral asset. However, if a collateral token with the same policy id but a new token name is added to the UTxO, the UTxO gets bigger and a bigger min Ada is needed for the ledger to accept such a UTxO. As a result, it might not be possible to add collateral to such loans if the min Ada in the UTxO can not cover that, s.a. in cases when it is chosen as the minimum value that satisfies the min Ada requirements.

Recommendation

I suggest allowing Ada increases in any checks that compare values. In particular, this finding is about the checks in the `validate_output_to_loan_for_adding_collateral` function.

Resolution

The issue was fixed in the commit `4fda819d4b` for the mentioned function.

FTL3-207 No liquidation discount

Category	Vulnerable commit	Severity	Status
Design Issue	`c9ad2a4960`	MEDIUM	RESOLVED

Description

In case of partial liquidation, the lender computes the remaining debt, the current collateral value and needs to return the rest of the value to the borrower after accounting for the remaining debt. There is no liquidation discount of any kind which would cover his exchange fees, potential last-minute token price changes, etc. That means that even upon a successful partial liquidation, the lender is not guaranteed to get away with enough tokens to cover the whole remaining debt.

Recommendation

Similar to the late repayment penalty, I suggest adding a liquidation discount parameter. It can even be variable and set by the parties as well. The goal is to allow the lender to claim a slightly larger portion of the pot, helping ensure they can cover the remaining debt even in the face of market volatility and transaction costs.

Resolution

The issue was fixed by the commit `4b1ee3fe68`. A `partialLiquidationPenaltyPerMille` variable was added. In partial liquidations, it further reduces the equity repaid to the borrower by this portion of the remaining debt, effectively acting as a liquidation discount.

FTL3-208 User stake credentials to authorize programmable token transfers

Category	Vulnerable commit	Severity	Status
Design Issue	`db29a2fe1d`	MEDIUM	RESOLVED

Description

User stake credentials are used directly as "ownership credentials" for programmable token outputs in repayment and borrower compensation scenarios. The `get_outputs_to_smart_credential` function then allows outputs at the programmable script payment credential with the user staking credential. That means that for the transfers of such tokens to be successful, the user needs to have complete control over the staking credential and the wallets building the transactions need to add the relevant signatures to the transaction.

It is somewhat unusual to use staking credential for this use case, as payment credentials are generally used to rule on spending eligibility, not the staking credentials.

Recommendation

I suggest using user payment credentials to note ownership in the programmable token scenarios.

Resolution

The issue was fixed in the commit `c23560b30f` .

FTL3-301 Permissioned lending party is chosen by an index out of context

Category	Vulnerable commit	Severity	Status
Logical Issue	`57f26cf730`	MINOR	RESOLVED

Description

There is an option for some loans to require an additional signing party. Any loan utilizing this lists whitelisted parties that can approve the lending action. In the code, the actual party which is verified to have signed the transaction is chosen from the list by an index that means something different in the context. It represents the order of the permissioned request input among the transaction inputs. While it is not impossible to construct a transaction that still validates as expected, I believe that constructing such transactions will be troublesome for transactions with multiple authorized parties and more permissioned requests in a single transaction.

Recommendation

I suggest either using different indices provided in the redeemer to determine the signing party for each permissioned request or simply checking that any party approves it.

Resolution

The issue was fixed in the commit `226e399697`. An index from the redeemer is used instead.

FTL3-302 Oracle's `valid_from` is unchecked

Category	Vulnerable commit	Severity	Status
Logical Issue	`610e4c46cd`	MINOR	RESOLVED

Description

Oracle feed contains the validity range in which it can be used. It is checked that the validity window is not too long by maintaining:

```
commonFeedData.valid_to - commonFeedData.valid_from <= constants.
   max_oracle_validity_range
```

It is then further checked, that the transaction's upper bound validity timestamp is before the oracle's `valid_to` timestamp.

However, the `valid_from` timestamp is unchecked. That means that the oracle's validity range might be invalid and still usable. Imagine having the oracle validity range set to `(valid_from, valid_to) = (now + 1000 years, now + 10 years)`. Such an artificially constructed invalid validity range would pass all the checks and such value could be used anytime in the coming 10 years.

The assumptions on exploiting this issue are quite big, it assumes rogue oracle signatories that would sign this invalid data off, hence it's just a minor severity issue. However, it is important to point out that the oracles could indeed publish data that could be valid effectively forever.

Recommendation

I suggest checking that the oracle's validity interval is not malformed and that the current transaction's validity interval lies within the oracle's validity interval:

```
oracle.valid_from < tx.valid_from < tx.valid_to < oracle.valid_to
```

Resolution

The issue was fixed in the commit `4fda819d4b` .

FTL3-303 Dutch auction can be bought before it starts

Category	Vulnerable commit	Severity	Status
Code Issue	`610e4c46cd`	MINOR	RESOLVED

Description

A dutch auction features a start date. However, it is not checked that it can not be bought before the start date. It actually just helps to calculate the current price. The collateral can be bought even before the start date. However, with a proportionally bigger price than the stated starting one.

Recommendation

I suggest either enforcing that the auction can not be interacted with before the start date, giving some time for the public to notice it; or renaming the "start" variables and clarifying that the auction is valid straight away.

Resolution

The issue was fixed in the commit `c49d69ef8c` . The auction can not be interacted with before the start date.

FTL3-304 Indexing repayments in repayment minting policy is troublesome

Category	Vulnerable commit	Severity	Status
Logical Issue	`c49d69ef8c`	MINOR	RESOLVED

Description

In the repayment minting policy, it is checked that only necessary repayment tokens are minted and that they go to the correct repayment outputs. To do that, all the loan inputs from the transaction are taken (whose order is lexicographical based on their output reference), those that do not result in a repayment output being created are skipped and those that do are checked. However, the `index` that is used to index the repayment outputs is the same one that corresponds to the loan input's index. It does not consider those loans that are skipped.

For example, let's consider 3 loan inputs. The first and the third result in repayment outputs. The second one doesn't. When processing the third loan input, the code would validate the third repayment output. However, there are just two of them as just two of the loan inputs should result in repayment outputs.

While it is theoretically feasible to create a dummy second repayment output to maintain the logic, it is troublesome, some min Ada would be locked, and the transactions would be unnecessary bigger.

Recommendation

I suggest first filtering out those loan inputs that should be considered for repayment token minting and just then indexing and validating them with the repayment outputs in that order.

Resolution

The issue was fixed in the commit `4fda819d4b`. It was then reintroduced in the commit `c69a7c268b`. Finally, the loan validator was split into 4 by the redeemer used, only one redeemer for all loan inputs is now enforced and thus the issue is not present anymore at commit `db29a2fe1d`.

FTL3-305 Burning and minting request and pool tokens is inconvenient

Category	Vulnerable commit	Severity	Status
Logical Issue	`dc3f840801`	MINOR	RESOLVED

Description

Similar to the issue FTL3-304, the request and pool minting policies both iterate over their mint value records. They are sorted lexicographically by the asset names there. The code skips burns and validates mints. However, the index is incremented even for those burned tokens. The index is used to index request or pool outputs where the minted token should go. That means that if there is a mint, a burn and a mint, in this order, the policy would need three request/pool outputs to succeed. Although it is not unfeasible to create an additional dummy request/pool, it is certainly inconvenient.

Recommendation

I suggest iterating on only the minted tokens in the request and pool minting policies, filtering out the burning quantities beforehand.

Resolution

The issue was fixed in the commit `4fda819d4b` .

FTL3-306 The `principalLTV` variable is over-used

Category	Vulnerable commit	Severity	Status
Code Issue	`c9ad2a4960`	MINOR	RESOLVED

Description

The variable is mandatory in all request and pool datums. It is hard to say something is a loan-to-value (LTV) when oracles are not used. In those cases, the variable even contains a different value. Whereas a typical LTV value is less than `1`, for cases when no oracles are used, the value bears the meaning of the "amount of collateral per single principal unit" — which should be more than `1` as loans are overcollateralized.

Even though it is strictly a semantics thing and as long as all parties are in-sync with the meaning, there's no impact, it sure can make mistakes easier.

Recommendation

I suggest remaking the field. The purpose of it is different in requests compared to pools. And it is different when using oracles compared to not using them. Make the guarantees easily readable from the structure to avoid making mistakes. For example for requests, you could have a `minPrincipal = Exactly principalAmount | Dynamic ltv`.

Resolution

The issue was fixed by the commit `e20d6c5ddf`. The variables were renamed to `minPrincipal` and `minCollateral`. They still refer to either the LTV-style ratio based on which the relevant minimum is computed or a multiplier that is used when oracles are not used.

FTL3-307 Ada oracle use is inconsistent

Category	Vulnerable commit	Severity	Status
Code Issue	`c9ad2a4960`	MINOR	RESOLVED

Description

All oracle feeds used in the protocol supply the price of a token to Ada. As such, Ada does not need an oracle, its value to Ada is clear. The handling of this edge case is inconsistent in the protocol. If Ada is the principal asset, a new logic was introduced that hardcoded the value. If Ada is the collateral asset, though, there's no such handling.

There might be no impact coming out of this as it's possible to create an oracle supplying the 1:1 value feed. For consistency and code cleanliness sake, it is worth unifying, though.

Recommendation

I suggest unifying the handling and, if necessary, extracting the hardcoded logic into a neat function so it's not repeated across 6 places in the code.

Resolution

The issue was fixed in the commit `a10fb17476`. The Ada oracle is now hardcoded in a single place, in the `retrieve_oracle_price` function.

FTL3-308 AMM formulas are based on a rational number that is then rounded

Category	Vulnerable commit	Severity	Status
Code Issue	`c9ad2a4960`	MINOR	RESOLVED

Description

This issue is specifically about the variables `wanted_token_a_amount` and `selling_token_a_amount`. They are rational numbers and their rational part is truncated inside the AMM functions. That is not always the correct way to handle them, though. The way they should be rounded depends on the context; is the function computing a lower bound or an upper bound? From a purely semantic standpoint, it is also not correct — if I need 5.9 tokens, it is not enough to compute how much tokens B I need to purchase 5 tokens.

Recommendation

As the rounding depends on the use case, I suggest making the argument an integer and thus delegating the rounding decision to the function that calls it.

Resolution

The issue was fixed by the commit `4b1ee3fe68` by the functions being removed as the `Pooled` oracle data type was phased out.

FTL3-309 Oracle safe-guards suggestion

Category	Vulnerable commit	Severity	Status
Design Issue	c9ad2a4960	MINOR	ACKNOWLEDGED

Description

Oracles either provide a fixed value determined by a CEX exchange rate, a price computed by a special method via dedicated oracles or a constant product liquidity pool's reserves. Not all the methods might be suitable for computing the exchange rate for all quantities, though. For example, if a small liquidity pool is used to determine the collateral's value in lovelace, it might look as-if it is quite worthless and thus ready to be liquidated. On the other side of the spectrum, using a fixed CEX exchange rate for a principal token on a big loan might also be misleading as there might be no demand to buy at that price that much and the price would dive significantly. It depends a lot on the tokens used, their liquidity, the size of the loans vs the liquidity source used for oracle feeds.

The severity of this issue is set as minor as the impact depends on the actual usage of feeds and loan sizes. Most data feeds need to be signed off by multiple parties and the mechanism for deciding which token uses which feed is not part of this audit. This is not to say that the results of this issue can not be catastrophical if unsuitable non-liquid feeds are used for too big loans.

Recommendation

There's no simple way to solve all of these issues and avoid all oracle manipulation. However, it is crucial to understand the issues behind the source of data. The oracle data might be technically correct, but economically off. I suggest limiting the size of the loans where certain oracle feeds can be used. This might be a separate field supplied by the oracle providers while still part of the feed. For CEX data, it can observe the order book depth where the price is more or less constant. For DEX feeds, it could be a more or less a slippage free zone. For bigger loans, the user might request a dedicated oracle type that would aggregate multiple liquidity sources or use an aggregated Charli3 oracle which should do this as well.

Resolution

The issue was acknowledged. The client plans to use Charli3 and highly liquid assets.

FTL3-310 Equity computation charges conversion fees to the lender

Category	Vulnerable commit	Severity	Status
Logical Issue	`c9ad2a4960`	MINOR	RESOLVED

Description

If the collateral crosses the dangerous liquidation threshold in the partial liquidation scenario, the remaining debt is computed and it is enforced that the remaining value from the collateral value is returned to the borrower. In the case of sourcing the oracle data from liquidity pools, the fees are charged to the lender. It is computed in such a way to determine the amount of principal tokens that are needed to convert to a fixed remaining collateral value in lovelace. That protects the borrower. However, it also means that the lender doesn't get the whole remaining debt out of the liquidation.

Recommendation

I suggest using a simple conversion method for equity computation in the pooled oracle feed instead of using the `token_b_received_from_selling_token_a_in_AMM_pool` function. Additionally, when rounding the equity amount, I suggest flooring it.

Resolution

The issue was fixed by the commit `8d0e95f6bc` . The pooled oracle type was retired and the value is floored now.

FTL3-311 Pool KYC token signature can be reused to borrow more

Category	Vulnerable commit	Severity	Status
Design Issue	`c9ad2a4960`	MINOR	RESOLVED

Description

The current design of the pool KYC token allows it to be reused across multiple transactions. Although the signed data includes the amount from the `Borrow` redeemer, it does not uniquely bind it to a single borrow action. As a result, the same user can reuse it to authorize borrowing more than the signed-off quantity. It is unclear how important it is to control the amount borrowed, though.

Recommendation

First, clarify whether limiting the borrowed amount is important. If it is, the signature should be bound to the specific borrowing instance. The simplest approach would be to include the output reference inside the signed data (noting that this may cause contention). Alternatively, if limiting the amount is not a requirement, sign only the relevant parts of the action, such as the borrower address and pool id.

Resolution

The issue was fixed by the commit `8d0e95f6bc` .

FTL3-312 Unlimited recasts do not work

Category	Vulnerable commit	Severity	Status
Documentation	`c9ad2a4960`	MINOR	RESOLVED

Description

The documentation on the `max_possible_recasts` field notes that negative number stands for unlimited recasts. However, the `is_recasting_permitted` function simply compares `doneRecasts < max_possible_recasts` and thus does not allow any recasts for a negative value of maximum recasts.

Recommendation

I suggest updating the documentation to not mention negative values and instead supply a big integer maximum to achieve near-unlimited recasts.

Resolution

The issue was fixed in the commit `a10fb17476` by updating the documentation. For seamingly unlimited recasts, the `max_possible_recasts` field can be set to a big integer.

FTL3-313 Borrowers can avoid late repayment penalty

Category	Vulnerable commit	Severity	Status
Logical Issue	`c9ad2a4960`	MINOR	RESOLVED

Description

There is a late repayment penalty for borrowers that don't repay in time but repay. The transaction's `validFrom` timestamp is compared with the time when the installment was supposed to be repaid. However, the borrower has incentive to push the transaction validity from timestamp as far into the past as possible to make it look as-if he was not late. And he can do that.

Recommendation

I suggest using the transaction validity range's upper bound in `Repay` and `Recast` transactions.

Resolution

The issue was fixed by the commit `8d0e95f6bc` .

FTL3-314 Installment amounts might not add up to the total principal and interest

Category	Vulnerable commit	Severity	Status
Logical Issue	c9ad2a4960	MINOR	ACKNOWLEDGED

Description

Aside from considering potential late penalties, there might be off by one errors in every installment totalling in a different sum to be repaid. The installemnts are computed and rounded up. The last repayment is computed the same way. That might be more in total than if there were a single repayment.

Recommendation

I think this can be handled well by an intuitive UX showing these. If you want to be precise I suggest flooring down individual repayment amounts and computing the last repayment as the total assumed loan plus interest minus total repaid. That might be a bit harder to compute for the amortization formula scenario with recasts. However, it is possible with a few additional fields.

Resolution

The issue was acknowledged.

FTL3-315 Total installments field for perpetual loans

Category	Vulnerable commit	Severity	Status
Code Issue	`c9ad2a4960`	MINOR	RESOLVED

Description

Every loan has a mandatory field `totalInstallments`. In case of perpetual loans, this does not make any sense, though. More than that, setting the value is actually dangerous because it is used even in perpetual loan scenarios inside the loan validator to determine whether an ongoing loan should be created or not. As a result, setting the field incorrectly for perpetual loan scenarios that should not have this field, might help the borrower claim his collateral without ever repaying the principal of the loan.

Recommendation

I suggest having the field only in cases where it makes sense, not for perpetual loans.

Resolution

The danger of the issue was mitigated by the commit `4b1ee3fe68`. The `totalInstallments` field is still present for all loans, however, it's not used in perpetual loan scenarios.

FTL3-316 Hash function mismatch in oracle key verification

Category	Vulnerable commit	Severity	Status
Logical Issue	`aef2d5dd0a`	MINOR	RESOLVED

Description

The pool KYC token validator uses `blake2b_256` to hash oracle verification keys but compares them against `VerificationKeyHash`-type values which are `blake2b_224` hashes on Cardano. This means that if real verification key hashes are put there, the comparison will always fail since the contract is comparing 32-byte hashes against expected 28-byte hashes. While Aiken allows 32-byte hashes to be put into the 28-byte type, it requires rehashing and saving the verification keys by `blake2b_256` for the contract to work properly which doesn't look expected.

Recommendation

I suggest changing the line #115 in the `pool_kyc_token.ak` file from `blake2b_256(compliance.token.oracle_key)` to `blake2b_224(compliance.token.oracle_key)`

Alternatively, if the `blake2b_256` usage is intentional, update the `whitelisted_oracles` type to explicitly use `Hash<Blake2b_256, VerificationKey>` to make it clear.

Resolution

The issue was fixed in the commit `51be906483` .

FTL3-317 Native tokens can be sent to programmable credential

Category	Vulnerable commit	Severity	Status
Logical Issue	`aef2d5dd0a`	MINOR	ACKNOWLEDGED

Description

The protocol works well with both native tokens and CIP-113 programmable assets. It handles both generically. A consequence of this, however, is that it is not checked which tokens go to which credential. Programmable token's spending credential ensures that its tokens do not go to a non-programmable address.

In contrast, nothing is enforced for native tokens. They might be sent to a programmable credential. It should be possible to retrieve them from the credential easily, by supplying a proof that the native token is not programmable. However, it might cause inconvenience to the user if e.g. the borrower's native tokens are sent to the programmable token's credential by the lender and now the borrower needs to use a specific wallet to retrieve them.

Recommendation

Consider implementing a validation that would enforce that a native token does not go to the programmable tokens' credential if there are no other programmable tokens added. To do that, you might parse the programmable token's redeemer to check whether the tokens are programmable or not. Finally, you could also remember this data for the two tokens used — the principal and the collateral.

Resolution

The issue was acknowledged as it is possible to retrieve the native tokens from the programmable credential.

FTL3-318 Big bond reference inputs can cause DoS via transaction limits

Category	Vulnerable commit	Severity	Status
Design Issue	`db29a2fe1d`	MINOR	RESOLVED

Description

Validators require bond reference inputs for address extraction, but attackers can weaponize these by creating bond outputs with excessive numbers of different tokens. When these bloated bond outputs are used as reference inputs, they might cause transactions to hit size limits, or to exceed execution units.

Impact

- **Lender DoS:** Malicious lenders can pack their bonds to make repayment/recast transactions fail.
- **Borrower DoS:** Malicious borrowers can pack their bonds to make liquidation transactions fail.

This can disrupt protocol operations and potentially force users into unfavorable situations.

Recommendation

Test that all affected transaction paths remain feasible even when bond reference inputs contain the maximum number of different tokens allowed by the ledger. If it is not feasible, different design choices might have to be taken.

Resolution

The issue was resolved by the client performing tests to show that the validators are optimized enough to handle very big bond reference inputs properly.

FTL3-401 Dropping a byte of a hash result is discouraged

Category	Vulnerable commit	Severity	Status
Logical Issue	`dc3f840801`	INFORMATIONAL	RESOLVED

Description

Hash functions are heavily studied and formally proven to satisfy multiple properties. Dropping a byte of the hashing result does not necessarily satisfy similar pseudo-random properties and is not generally studied that much. It is always a better idea to use a shorter but full hash result of a studied function and append any additional data to that.

Recommendation

I suggest reworking request and pool tokens' asset name generation to use a shorter hash function s.a. `blake2b_224` instead of shortening the `sha2_256`'s result.

Resolution

The issue was fixed by the commit `9e7adc4470`.

FTL3-402 Request id and pool id might be identical

Category	Vulnerable commit	Severity	Status
Logical Issue	`c9ad2a4960`	INFORMATIONAL	RESOLVED

Description

Even though a special care is taken to ensure that no multiple equal request tokens and no multiple pool tokens can be minted, if request tokens are minted in the same transaction as pool tokens, it might happen that the same asset names are assigned. As only the asset name is used as a form of identifier in a loan field called `originAssetName`, they might conflict there. The field is not really used in the on-chain, but it might be used in the off-chain.

Recommendation

I suggest renaming the field to `originId` and prefixing the asset name by its origin, a request or a pool, to really yield unique identifiers.

Resolution

The issue was fixed in the commit `a10fb17476` by renaming the field and prefixing it by its script origin.

FTL3-403 Equity payment is in the principal asset

Category	Vulnerable commit	Severity	Status
Design Issue	c9ad2a4960	INFORMATIONAL	ACKNOWLEDGED

Description

It is more common to repay the equity in the collateral asset instead of the principal asset that this protocol repays the equity in. For example, if a borrower gets liquidated when supplying ADA as collateral to borrow a stablecoin, his collateral is taken and the leftover value is returned in the stablecoin. It is somewhat counterintuitive. It might make more sense to do when assuming the possibility of NFT collaterals. However, it is applied for all loans.

Recommendation

I suggest reconsidering the asset in which the equity is paid, potentially even making it possible for the parties to choose whichever they prefer.

Resolution

The issue was acknowledged.

FTL3-404 Code quality, naming, and documentation issues

Category	Vulnerable commit	Severity	Status
Code Style	`c23560b30f`	INFORMATIONAL	RESOLVED

Description

The following code quality, naming, and documentation issues have been identified:

- **Dead Code:**
  - `address_in_signatures` — Unused function that should be removed.
  - `is_output_delegated_to_sc` — Unused function that should be removed.
- **Naming Issues:**
  - `validate_output_to_loan_for_changing_collateral` — Function parameters `loanOutput` and `loanValue` refer to different loan UTxOs, suggesting rename of `loanValue` to `oldLoanValue`.
  - Orcfax `fluid_price` field should be renamed to e.g. `token_price` for general accuracy.
  - Orcfax `Rational` datum type potentially shadows Aiken's built-in `Rational` type, suggesting renaming it to e.g. `OrcfaxRational`.
- **Code Style:**
  - Orcfax `feed_id` validation uses `starts_with` prefix matching instead of exact comparison.
  - Orcfax price comparison unnecessarily reduces the prices in the code; comparing unreduced rationals is possible.
- **Documentation:**
  - "Liquidation for too low LTV" comment describes the opposite scenario.
  - Outdated oracle documentation: "Validate that n/m oracles have signed this tx" no longer accurate.
  - Oracle config documentation as well as implementation mentions 7-hour price tolerance while `oracle.ak` mentions 6-hour tolerance.

Recommendation

Address these issues to improve code maintainability, reduce confusion, and ensure documentation accuracy.

Resolution

All issues were resolved by the commit `c7b9d1cd2d`, except for one code style issue that is present in Orcfax-made code which the team just copied.

A Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the agreement between VacuumLabs Bohemia s.r.o. (Vacuumlabs) and FT Labs GmbH (Client) (the Agreement), or the scope of services, and terms and conditions provided to the Client in connection with the Agreement, and shall be used only subject to and to the extent permitted by such terms and conditions. This report may not be transmitted, disclosed, referred to, modified by, or relied upon by any person for any purposes without Vacuumlabs's prior written consent.

This report is not, nor should be considered, an endorsement, approval or disapproval of any particular project, team, code, technology, asset or anything else. This report is not, nor should be considered, an indication of the economics or value of any technology, product or asset created by any team or project that contracts Vacuumlabs to perform a smart contract assessment. This report does not provide any warranty or guarantee regarding the quality or nature of the technology analysed, nor does it provide any indication of the technology's proprietors, business, business model or legal compliance.

To the fullest extent permitted by law, Vacuumlabs disclaims all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. This report is provided on an as-is, where-is, and as-available basis. Vacuumlabs does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by Client or any third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services, assets and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Vacuumlabs will not be a party to or in any way be responsible for monitoring any transaction between you and client and/or any third-party providers of products or services.

This report should not be used in any way by anyone to make decisions around investment or involvement with any particular project, services or assets, especially not to make decisions to buy or sell any assets or products. This report provides general information and is not tailored to anyone's specific situation, its content, access, and/or usage thereof, including any associated services or materials, shall not be considered or

relied upon as any form of financial, investment, tax, legal, regulatory, or other advice.

This report is based on the scope of materials and documentation provided for a limited review at the time provided. Vacuumlabs prepared this report as an informational exercise documenting the due diligence involved in the course of development of the Client's smart contract only, and THIS REPORT MAKES NO CLAIMS OR GUARANTEES CONCERNING THE SMART CONTRACT'S OPERATION ON DEPLOYMENT OR POST-DEPLOYMENT. This report provides no opinion or guarantee on the security of the code, smart contracts, project, the related assets or anything else at the time of deployment or post deployment. Smart contracts can be invoked by anyone on the internet and as such carry substantial risk. VACUUMLABS HAS NO DUTY TO MONITOR CLIENT'S OPERATION OF THE PROJECT AND UPDATE THE REPORT ACCORDINGLY.

THE INFORMATION CONTAINED IN THIS REPORT MAY NOT BE COMPLETE NOR INCLUSIVE OF ALL VULNERABILITIES. This report is not comprehensive in scope, it excludes a number of components critical to the correct operation of this system. You agree that your access to and/or use of, including but not limited to, any associated services, products, protocols, platforms, content, assets, and materials will be at your sole risk. On its own, it cannot be considered a sufficient assessment of the correctness of the code or any technology. This report represents an extensive assessing process intending to help Client increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology, however blockchain technology and cryptographic assets present a high level of ongoing risk, including but not limited to unknown risks and flaws.

While Vacuumlabs has conducted an analysis to the best of its ability, it is Vacuumlabs's recommendation to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring and/or other auditing and monitoring in line with the industry best practice. The possibility of human error in the manual review process is highly real, and Vacuumlabs recommends seeking multiple independent opinions on any claims which impact any functioning of the code, project, smart contracts, systems, technology or involvement of any funds or assets. VACUUMLABS'S POSITION IS THAT EACH COMPANY AND INDIVIDUAL ARE RESPONSIBLE FOR THEIR OWN DUE DILIGENCE AND CONTINUOUS SECURITY.

B Audited files

The files and their hashes reflect the final state at commit `aed7d340119c56e8f8f02cbefcc53810ce7df0fc` after all the fixes have been implemented.

SHA256 hash	Filename
97605...83634	lib/fluidtokens/authorizer.ak
9f9d5...34bc2	lib/fluidtokens/constants.ak
e4942...02447	lib/fluidtokens/finance.ak
e1368...7468c	lib/fluidtokens/oracle.ak
b5dd0...e2a39	lib/fluidtokens/types/config.ak
952b9...423c7	lib/fluidtokens/types/dutch_auction.ak
5f6ba...d25c3	lib/fluidtokens/types/general.ak
2487b...ba35d	lib/fluidtokens/types/loan.ak
d99ae...77a8d	lib/fluidtokens/types/oracle.ak
13e9d...7d645	lib/fluidtokens/types/pool.ak
db580...e5a1b	lib/fluidtokens/types/repayment.ak
0df35...f6907	lib/fluidtokens/types/request.ak
16929...cab58	lib/fluidtokens/utils.ak

Continued on next page

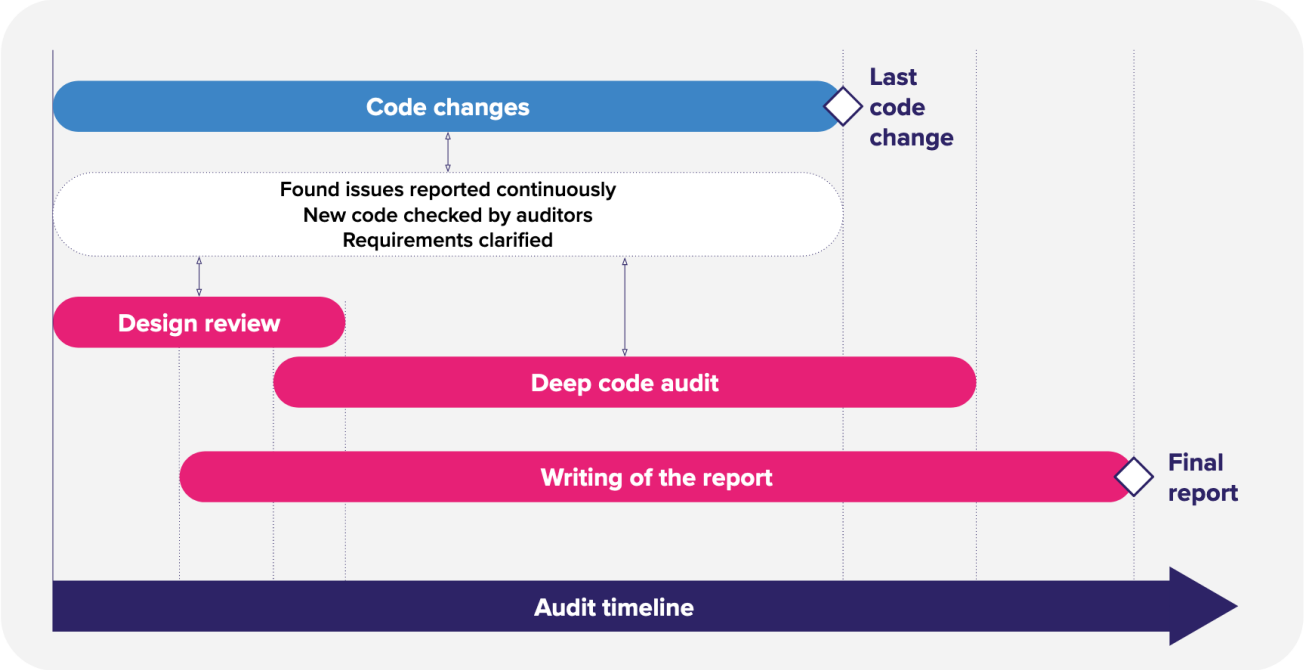
SHA256 hash	Filename
3aed1...e438a	lib/smart-tokens/utils.ak
a0f53...2dfb3	validators/bond.ak
37a65...86b00	validators/conditions/pool_kyc_token.ak
17857...2ec42	validators/conditions/request_specific_lenders.ak
29548...3bcbf	validators/config.ak
eed40...a239b	validators/dutch_auction.ak
bd519...a564d	validators/general_spend.ak
74f21...fb2dc	validators/lender_smart_wallet.ak
9d815...f66e0	validators/loan_change_collateral_action.ak
d65be...57620	validators/loan_claim_action.ak
07278...fc2bc	validators/loan_recast_action.ak
0fb6d...e6c4c	validators/loan_repay_action.ak
d2c38...4f94d	validators/loan.ak
423d9...5d7e6	validators/oracle.ak
1e888...fbf38	validators/pool.ak
6b9ac...8de80	validators/repayment.ak
236c1...94a72	validators/request.ak

Please note that I did not audit Aiken itself, the underlying CIP-113 programmable token implementation or the external oracles that are used; they were assumed to function correctly and as designed.

C Methodology

Vacuumlabs' agile methodology for performing security audits consists of several key phases:

1. Design reviews form the initial stage of our audits. The goal of the design review is to find larger issues which result in large changes to the code fast.
2. During the deep code audit, we verify the correctness of the given code and scrutinize it for potential vulnerabilities. We also verify the client's fixes for all discovered vulnerabilities. We provide our clients with status reports on a continuous basis providing them a clear up-to-date status of all the issues found so far.
3. We conclude the audit by handing over a final audit report which contains descriptions and resolutions for all the identified vulnerabilities.



Throughout our entire audit process, we report issues as soon as they are found and verified. We communicate with the client for the duration of the whole audit. During our audits, we check several key properties of the code:

- Vulnerabilities in the code
- Adherence of the code to the documented business logic
- Potential issues in the design that are not vulnerabilities
- Code quality

During our manual audits, we focus on several types of attacks, including but not limited to:

1. Double satisfaction
2. Theft of funds
3. Violation of business requirements
4. Token uniqueness attacks
5. Faking timestamps
6. Locking funds indefinitely
7. Denial of service
8. Unauthorized minting
9. Loss of staking rewards

D Issue classification

Severity levels

The following table explains the different severities.

Severity	Impact
CRITICAL	Theft of user funds, permanent freezing of funds, protocol insolvency, etc.
MAJOR	Theft of unclaimed yield, permanent freezing of unclaimed yield, temporary freezing of funds, etc.
MEDIUM	Smart contract unable to operate, partial theft of funds/yield, etc.
MINOR	Contract fails to deliver promised returns, but does not lose user funds.
INFORMATIONAL	Best practices, code style, readability, documentation, etc.

Resolution status

The following table explains the different resolution statuses.

Resolution status	Description
RESOLVED	Fix applied.
PARTIALLY RESOLVED	Fix applied partially.
ACKNOWLEDGED	Acknowledged by the project to be fixed later or out of scope.
PENDING	Still waiting for a fix or an official response.

Categories of issues

The following table explains the different categories of issues.

Category	Description
Design Issue	High-level issues in the design. Often large in scope, requiring changes to the design or massive code changes to fix.
Logical Issue	Medium-sized issues, often in between the design and the implementation. The changes required in the design should be small-scaled (e.g. clarifying details), but they can affect the code significantly.
Code Issue	Small in size, fixable solely through the implementation. This category covers all sorts of bugs, deviations from specification, etc.
Code Style	Parts of the code that work properly but are possible sources of later issues (e.g. inconsistent naming, dead code).
Documentation	Small issues that relate to any part of the documentation (design specification, code documentation, or other audited documents). This category does not cover faulty design.
Optimization	Ideas on how to increase performance or decrease costs.

E Report revisions

This appendix contains the changelog of this report. Please note that the versions of the reports used here do not correspond with the audited application versions.

v1.0: Main audit

Revision date: 2025-08-27
Final commit: `aed7d340119c56e8f8f02cbefcc53810ce7df0fc`

We conducted the audit of the main application. To see the files audited, see Audited files.

Full report for this revision can be found at url.

F About us

Vacuumlabs has been building crypto projects since the early days.

- We helped create WingRiders, currently the second largest decentralized exchange on Cardano (based on TVL).
- We are behind the popular AdaLite wallet. It was later improved into a multichain wallet NuFi.
- We built the Cardano applications for the hardware wallets Ledger and Trezor.
- We built the first version of the cutting-edge decentralized NFT marketplace Jam On Bread on Cardano with truly unique features and superior speed of both the interface and transactions.

Our auditing team is chosen from the best.

- Talent from esteemed Cardano projects: WingRiders and NuFi.
- Rich experience across Google, traditional finance, trading and ethical hacking.
- Award-winning programmers from ACM ICPC, TopCoder and International Olympiad in Informatics.
- Driven by passion for program correctness, security, game theory and the blockchain technology.

Note: Vacuumlabs Auditing continues as Invariant0. See more here.

vacuumlabs

Contact us:
info@invariant0.com